Serial Interface Controller

Anand Kandasamy
Instrumentation Division
Brookhaven National Laboratory
Upton, NY 11973

April 14, 1995

1. Introduction:

The idea of building a Serial Interface Controller (SIC) proposed by **Paul O' Connor**, Instrumentation Division, BNL is to determine the feasibility of incorporating a Serial Interface Controlled CMOS IC's [1] for charge amplification, shaping, analog storage and multiplexing used in particle detectors for high energy physics experiments. The serial data pumped into the CMOS IC's will be used to control many circuit parameters like digitally controlled gain, shaping time, precision preamplifier calibration circuits and many other parameters like timing discriminators mode of operation.

The SIC board built will be tested on a Serial Interface Controlled Digital - to - Analog Convertor, which follows either Motorola's SPI/QSPI or National Semiconductors Microwire interface technique. The DAC chosen for this was MAXIM's MAX537, a Quad, 12-bit DAC. The function of this controller can be achieved by using some on-shelf micro-controllers like the Motorola's MC68HC11, which offers dedicated SPI ports. The drawback encountered in using this controller is the overhead involved in putting together an user interface where the user can dynamically change is settings and load the SIC device. This is very critical in testing fewer number of CMOS IC's having SIC. The SIC board described here takes care of this dynamic user interface issue.

1. P.O' Connor, Low Noise Signal Processing IC for Interpolating Cathode Strip Chambers, BNL-61085, (submitted to IEEE Transaction on Nuclear Science).

2. Description of an SIC device:

The SIC device described here is the MAX537, which is the device that has been chosen for testing the SIC board.

The MAX 537 combines four 12-bit, voltage output digital -to- analog converters (DAC) and four precision amplifiers. Each DAC has a double buffered input, organized as an input register followed by a DAC register. A 16-bit serial word is used to load data into each input/DAC register. The serial interface is compatible with either SPI/QSPI^{TM} or Microwire^{TM}, and allows the input and DAC registers to be independently or simultaneously with a single software command. The DAC registers can be simultaneously updated with a hardware $\overline{\text{LDAC}}$ pin. Figure 1. shows the functional diagram of MAX 537.

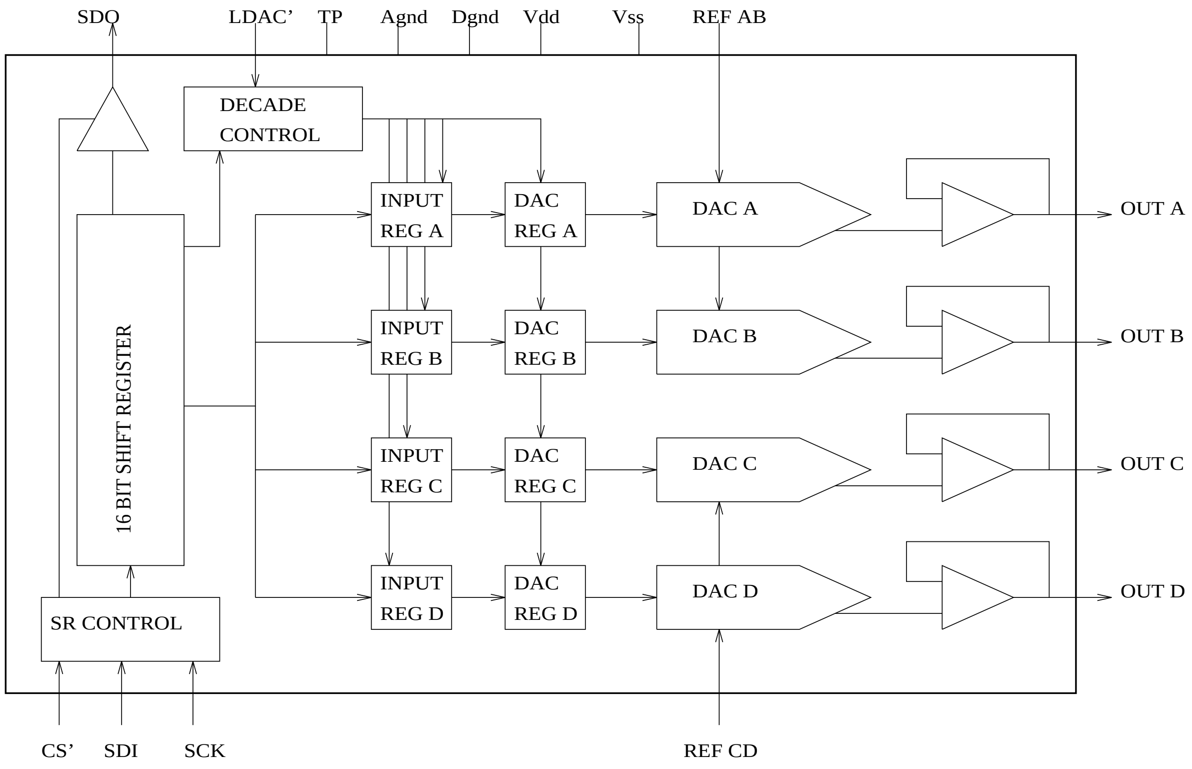
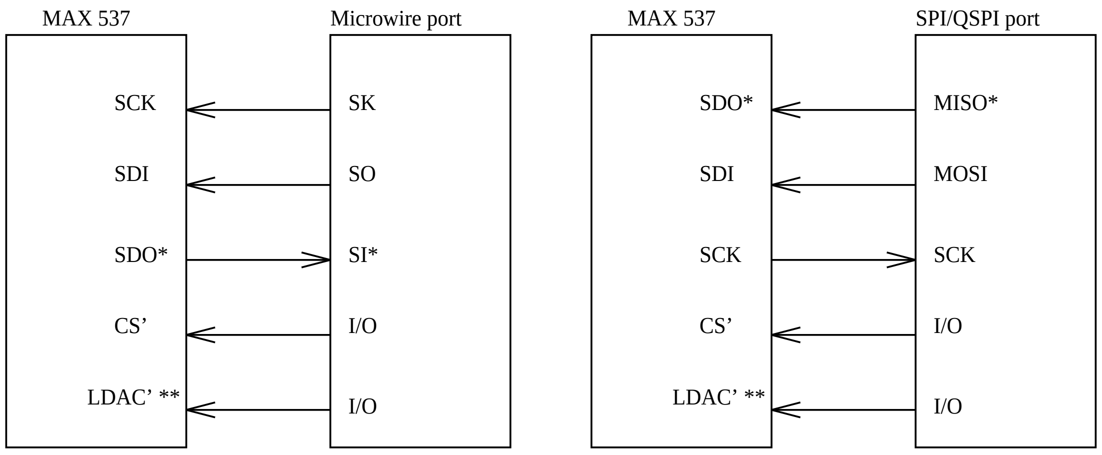


Figure.1 Functional Diagram of the MAX 537 DAC

• Serial-Interface Configuration:

The MAX 537's 3-wire or 4-wire serial interface is compatible with both Microwire and SPI/QSPI. The interface is shown in figure.2 In this configuration, $\overline{\text{LDAC}}$ can be tied either high or low for a 3-wire interface, or used as the fourth input with a 4-wire interface. The connection between SDO and the serial-interface port is not necessary, but may be used for data echo. With a 3-wire interface ($\overline{\text{CS}}$, SCK, SDI) and $\overline{\text{LDAC}}$ tied high, the DAC's are double buffered. In this mode depending on the command issued through the serial interface, the input register(s) may be loaded without affecting the DAC register(s), the DAC register(s) may be simultaneously updated from the input registers. With a 3-wire interface ($\overline{\text{CS}}$, SCK, SDI) and

$\overline{\text{LDAC}}$ tied low, the DAC registers remain transparent. Any time an input register is updated, the change will appear at the DAC output with the rising edge of $\overline{\text{CS}}$.



* SDO/SI conection is used for readback if needed
** LDAC' is not needed for a 3-wire interface

* SDO/MISO conection is used for readback if needed
** LDAC' is not needed for a 3-wire interface

Figure. 2. 3-wire and 4-wire configuration.

• Serial Interface Description:

The MAX 537 require 16 bits of serial data. Data is sent MSB first. ($\overline{\text{CS}}$ must remain low until all 16 bits are transferred) the serial data is composed of two DAC address bits (A1, A0),two control bits (C1, C0), and t he 12 data bits D11....D0 (figure 3.)

MSB					**LSB**
← **16 bits of serial Data** →					
Address Bits		**Control Bits**		**DATA BITS**	
				MSB	**LSB**
A1	**A0**	**C1**	**C0**	**D11**	**D0**
4 Addres/Control bits				**12 DATA bits**	

Figure. 3. Serial-Data Format (MSB sent first)

Figure 4 shows the serial interface timing diagram. The chip select pin ($\overline{CS}$) must be low to enable the DAC's serial interface. When $\overline{CS}$ is high, the interface control circuitry is disabled and the serial data output pin (SDO) is driven high. Data is clocked into the internal shift-register via the serial data in pin SDI) on SCK's rising edge. Data is latched into the appropriate input/DAC registers on $\overline{CS}$ rising edge.

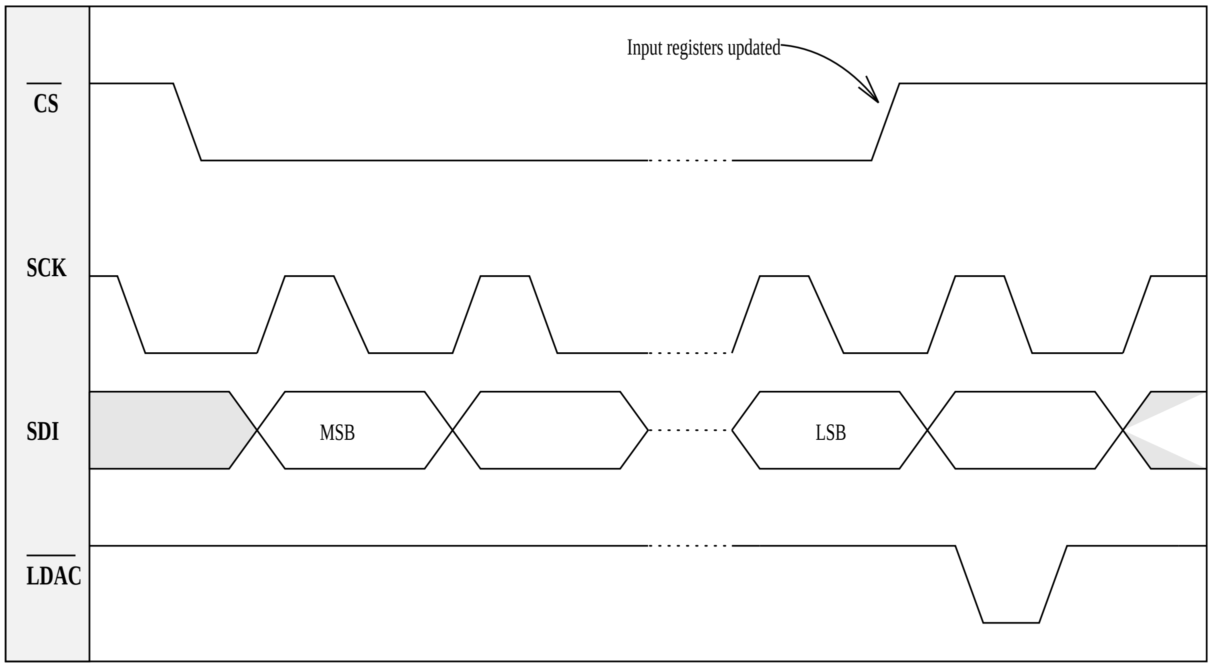


Figure 4. Serial-Interface Timing Diagram.

• **Serial Data Output:**

The serial data output SDO, is the internal shift registers output. The MAX 537 can be programmed so that data is clocked out of SDO on SCK's rising edge (MODE 0) or falling edge (MODE 1). On power-up SDO defaults to MODE 0 timing.

Table 1: Serial Interface Programming Commands.

A1	A0	C1	C0	D11.....D0	$\overline{\text{LDAC}}$	FUNCTION
0	0	0	1	12 Bit data	1	Load Dac A, Input register, O/P unchanged
0	1	0	1	12 Bit data	1	Load Dac B, Input register, O/P unchanged
1	0	0	1	12 Bit data	1	Load Dac C, Input register, O/P unchanged
1	1	0	1	12 Bit data		Load Dac D, Input register, O/P unchanged
0	0	1	1	12 Bit data	1	Load A input register, All Dac reg. updated
0	1	1	1	12 Bit data	1	Load B input register, All Dac reg. updated
1	0	1	1	12 Bit data	1	Load C input register, All Dac reg. updated
1	1	1	1	12 Bit data	1	Load D input register, All Dac reg. updated
X	0	0	0	12 Bit data	X	Load all DAC from shift register
X	1	0	0	XX........XX	X	No operation (NOP)
0	X	1	0	XX........XX	1	Update all DAC from their DAC register.
1	1	1	0	XX........XX	X	configure to MODE 0
1	0	1	0	XX........XX	X	configure to MODE 1
0	0	X	1	12 Bit data	0	Load Dac A and update Dac A immediately
0	1	X	1	12 Bit data	0	Load Dac B and update Dac B immediately
1	0	X	1	12 Bit data	0	Load Dac C and update Dac C immediately
1	1	X	1	12 Bit data	0	Load Dac D and update Dac D immediately

3. Serial Interface Controller Board Design:

• User Interface:

1. The user interfaces to the controller board by setting the digital code corresponding to the data bits D11....D0 of the MAX 537. The user input is in the form of a bank of switches, which are of Hexadecimal output type. Three switches are needed for each DAC.

2. A monostable multivibrator is used as the second user interface to initiate the serial data download sequence.

• Main controller:

The main controller does the task of reading in the 48 bits of information (12 bits/dac X 4) and when the user initiates the download sequence, it pumps out data serially along with two more signals SCK, the clock for the SIC device and $\overline{CS}$ for updating SIC device registers. The main controller has two modes of operation namely the Master and Slave mode. For configuring the MAX 537 the Master mode of operation is selected. In this mode the 48 data bits are read from the user interface and shifted out on demand. In the slave mode of operation 64 bit serial word is serially obtained from an external controller which fetches the 64 bit word stored in an external PROM or RAM. This mode is used for configuring larger number of SIC devices. This mode is introduced keeping in mind its use when the controller is used to configure SIC CMOS shaper IC's in a typical detector application where the device count exceeds 1000. Figure 5 shows the MODE 0 configuration and Figure 6 shows the MODE 1 configuration.

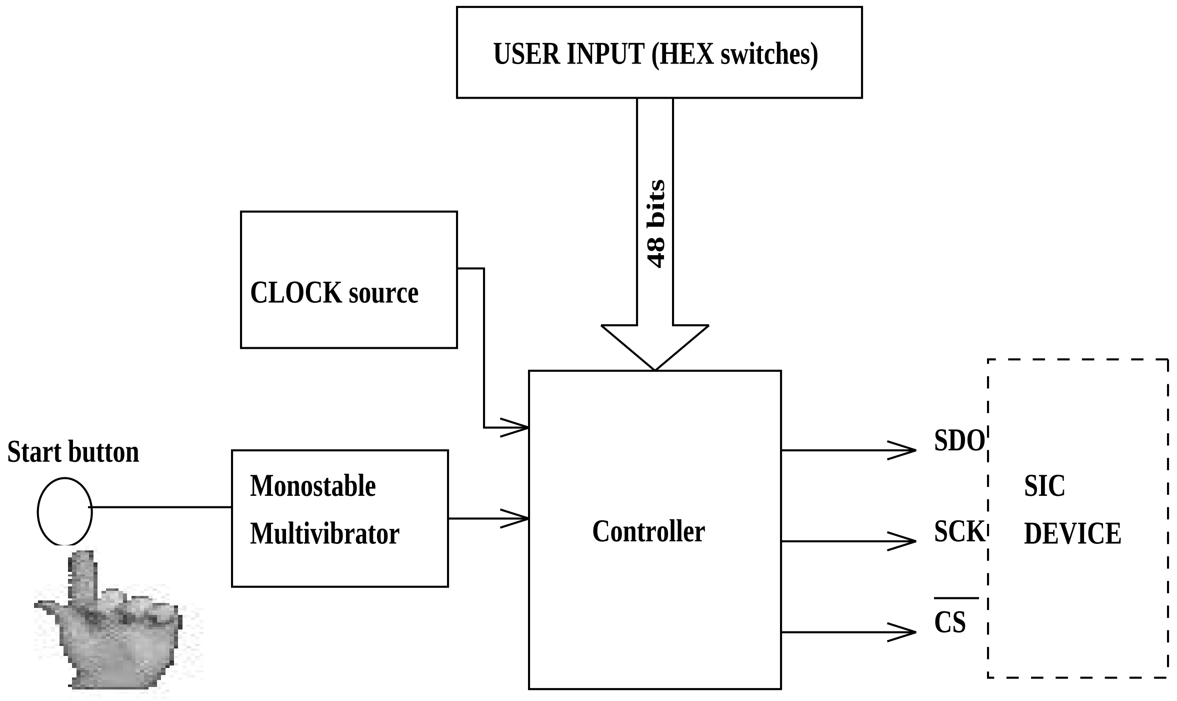


Figure 5. MODE 0 (Master) operation.

The SIC device in the above figure is the MAX 537 DAC. In the MODE 1 mode of operation the SIC device can be any SIC device including the SIC CMOS shaper IC's as mentioned before.

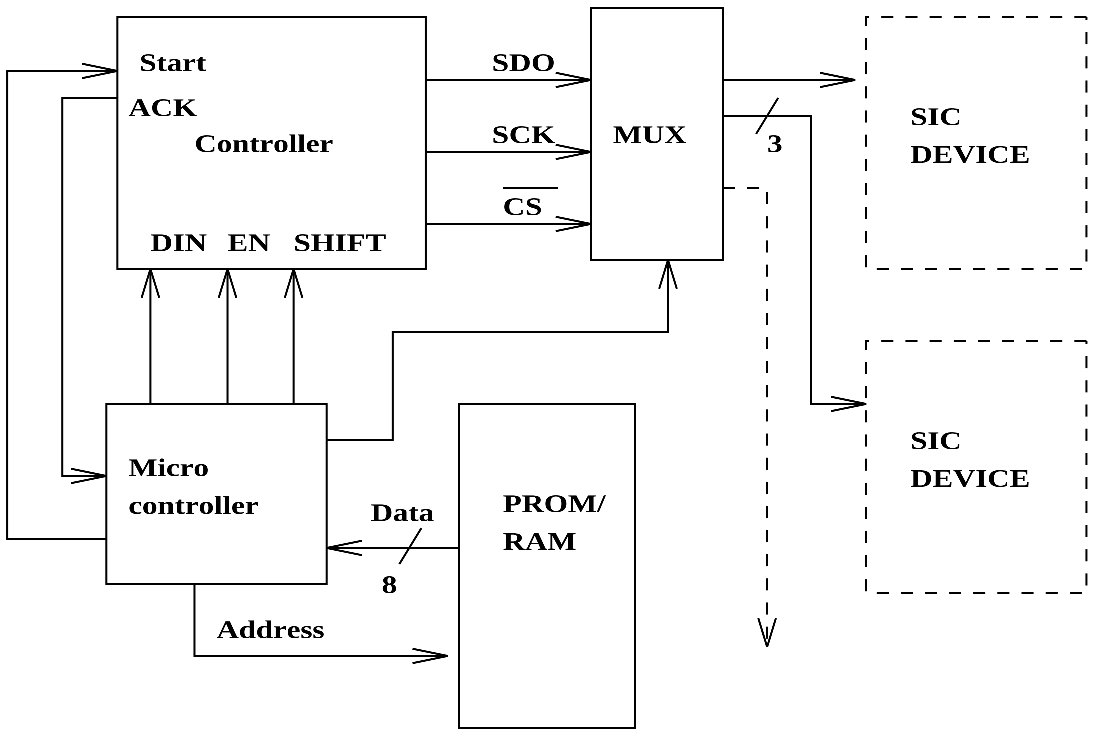


Figure 6. Mode 1 (Slave) operation.

4. Main controller design.

The Main controller for serial interface controller is realized using a XILINX FPGA. The device used is XC3064-70-PC84. This is an 84 pin PLCC device. The main factor influencing the choice of this device is the pin count, as the controller required 48 bits for the user data input alone.

The main controller contains the following sub-modules:

1. 64 BIT parallel loadable serial output shift register.
2. Counters (10 bit and 4 bit).
3. Internal clock divider and clock sequencer.
4. Control signals generating module.

and a few other logics for operating mode determination.

The operation of the main controller is as follows.

• MODE 0 (MASTER) mode:

1. When the user triggers the down load sequence start button, the pulse from the monostable multivibrator clocks a D-Flip-flop to logic ’1’ and this output is used to enable the internal clock divider and

sequencer module.

2. The mode of the external SIC device is assumed to be at MODE 0 (rising edge latching). This is assumed so that no additional steps are necessary to program the external SIC device to MODE 1 (falling edge latching).

3. One clock period is used for loading the 64 bit parallel loadable shift registers with the user input from the hexadecimal output switches.

4. 64 additional clock cycles are used to complete the entire down load sequence for all the 4 DAC's in the external SIC device (MAX 537).

5. At the end of every 16 internal clock cycles, $\overline{\text{CS}}$ signal is generated to update the input registers.

6. After the end of the entire download sequence the D-flip flop set to logic '1' by the trigger from the monostable multivibrator is reset and the entire controller waits for the next user trigger.

• MODE 1 (SLAVE) mode.

1. The Shiftenable line is held high by the external micro-controller.

2. The micro controller shifts in 64 bit of data into the serial input pin (DIN) of the main controller, the shift register in this mode uses an external signal from the micro-controller to shift the 64 bits of data.

3. At the end of every 15/16 bits of data shifted the main controller informs the main controller that the next bit is the 16th bit/acknowledges the receipt of 16 bits. This signal is useful for handshaking purposes.

4. At the end of the 64 bit shifting sequence, the Shiftenable is pulled low by the micro-controller.

5. The micro-controller initiates the download sequence through the trigger pin. On receiving this signal the main controller performs the same operation as that of mode 0, except it does not use on clock period for loading the user input.

6. After the end of the entire download sequence the D-flip flop set to logic '1' by the trigger from the micro-controller is reset and the entire controller waits for the next micro-controller trigger.

The schematic of the SIC controller is shown in figure 7. The shift register in the controller has its higher order 4 bits hard wired inside the LCA. These 4 bits corresponds to the 4 address and control bits necessary for programming the MAX 537. Detailed schematic of internal modules can be found in Appendix. A.

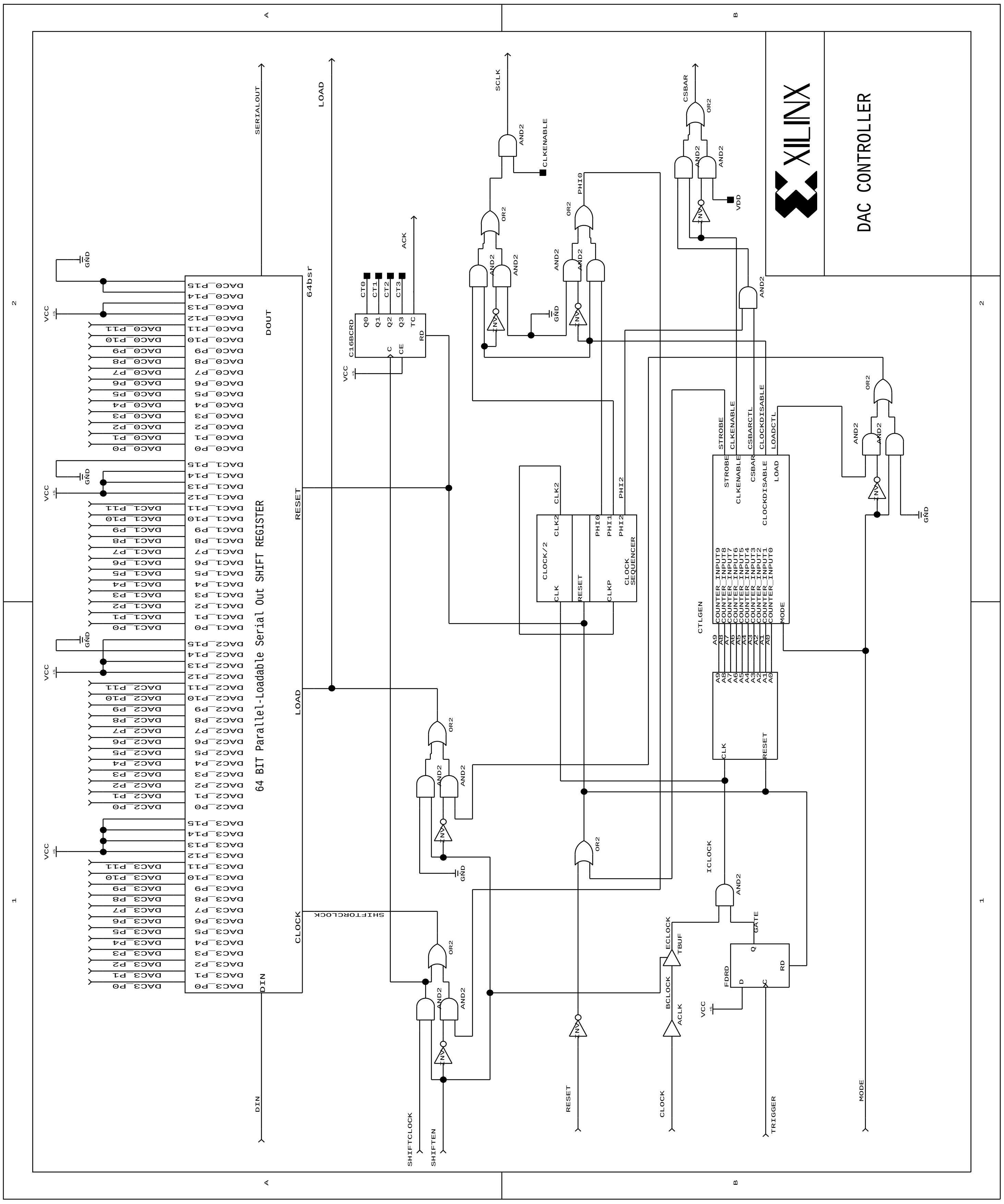


Figure 7. Schematic of the DAC controller.

The board clock of 10 MHz is divided internally by two relaxing the circuit timing parameters. The circuit shown in figure 8, known as glitch free sequencer generates three glitch free pulses corresponding to the successive half periods of the clock. This sequencer is an useful module since it generates pulses to be used for shifting the data in to the external SIC device and the clock for the external SIC device. The rising edge of the first pulse is used by the internal shift register to shift its data out of the SDO pin and the rising edge of the second pulse (delayed by an half period of the input clock) is used as the SCK (serial clock) for the external SIC device. As it can be seen due to the half period delay in the availability of the data at the external SIC device pin SDI and the clock for latching it, setup time for the flip-flops are relaxed.

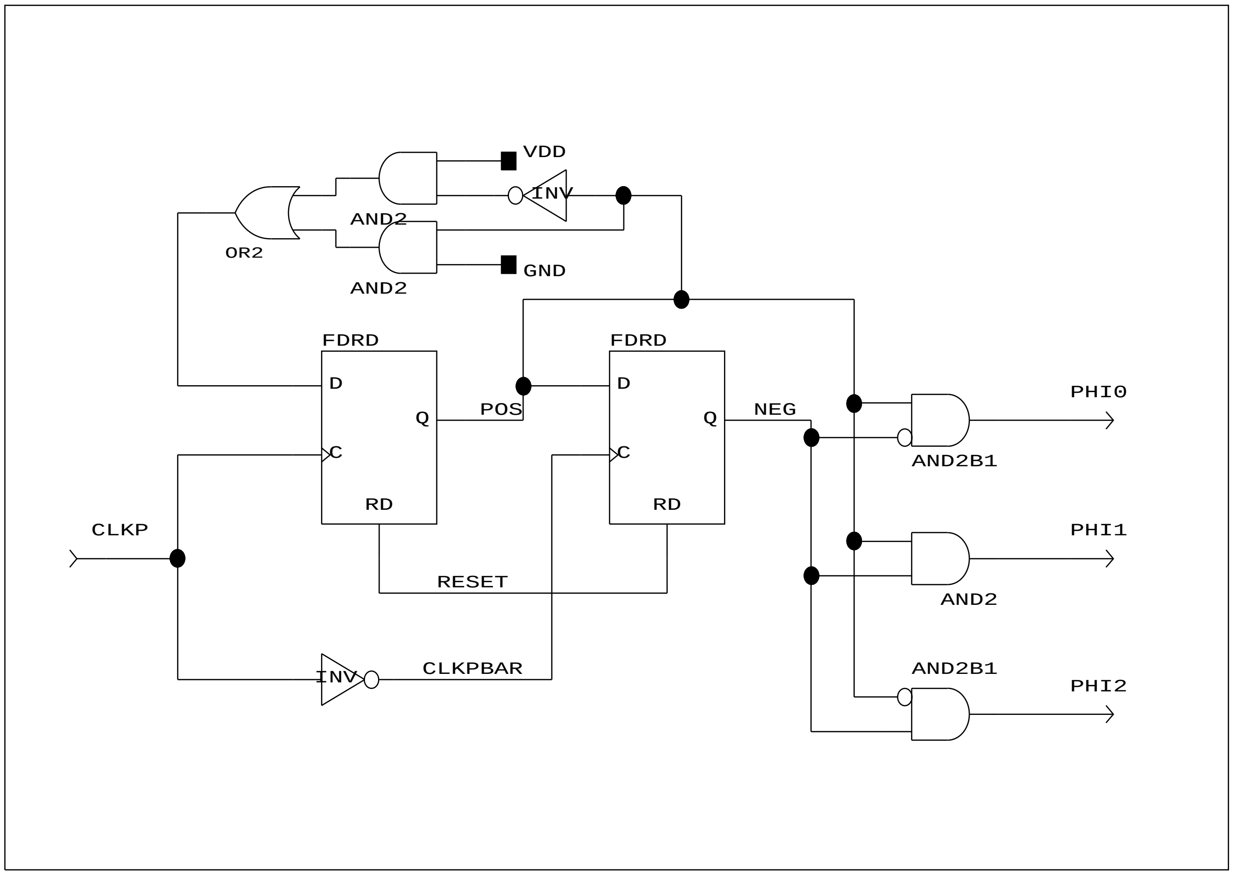


Figure 8. Glitch free clock sequencer.

The glitch free clock sequencer in the above figure reduces the clock frequency by 1/4 and also the duty cycle is now 25%. The signal phi0 is used as the internal 64 bit shift register shift and load clock, signal phi1 is used as the SCK clock for the external SIC device and the signal phi2 is used to aid in the removal of glitches in the $\overline{\text{CS}}$ due to race conditions. The waveform of the glitch free sequencer is shown in figure 9.

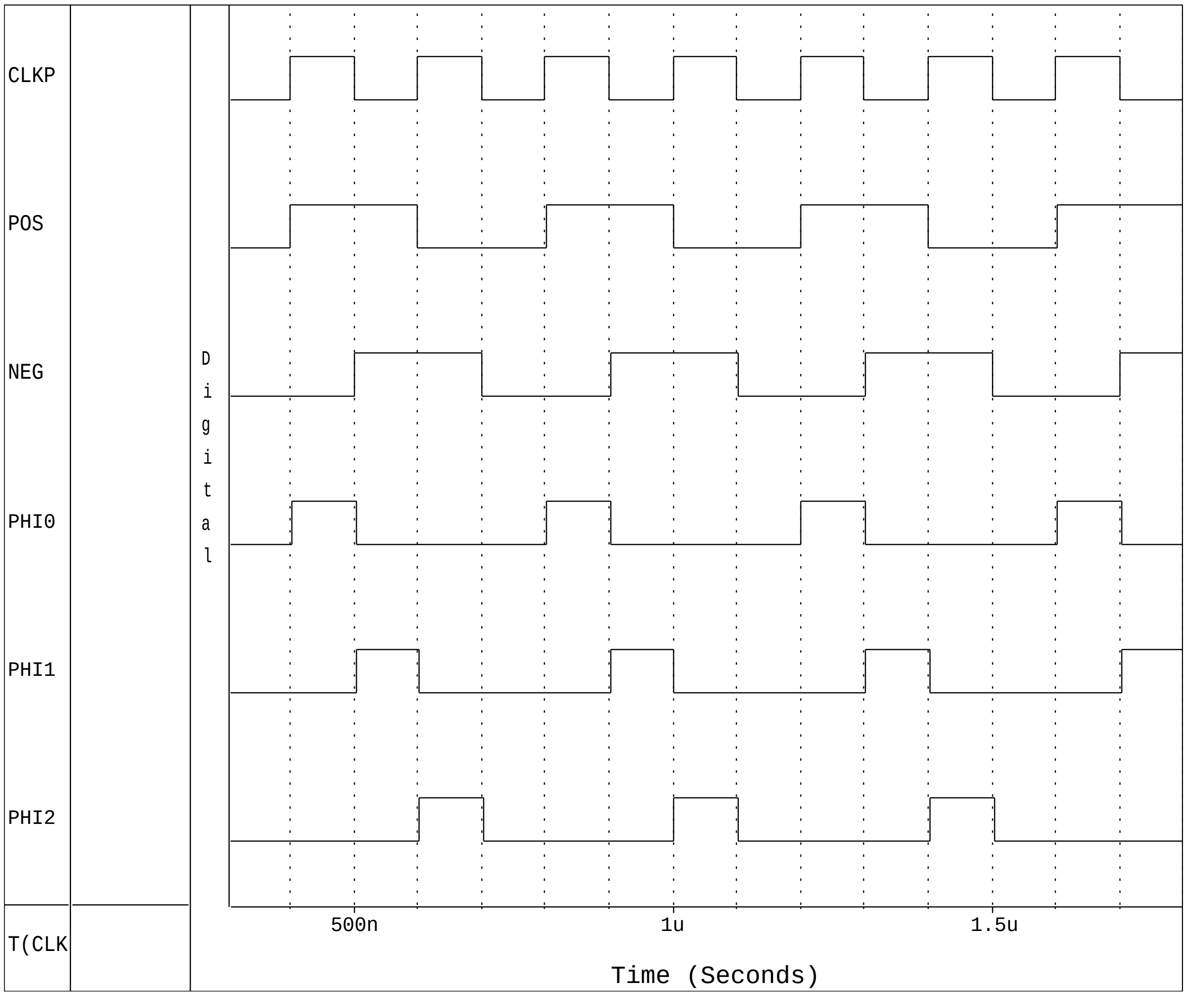


Figure 9. Waveforms for the glitch-free sequencer.

The waveforms for the MODE 0 (MASTER) operation of the DAC controller built using the XILINX FPGA is shown in figure 10. In the waveform, trigger is the monostable multivibrator output. clk2 is the divided by two clock of the master clock. The load clock being high for one clock cycle after the trigger can be noticed, during this clock edge, the user input from the hexa-decimal switches are loaded in to the 64 bit shift registers.

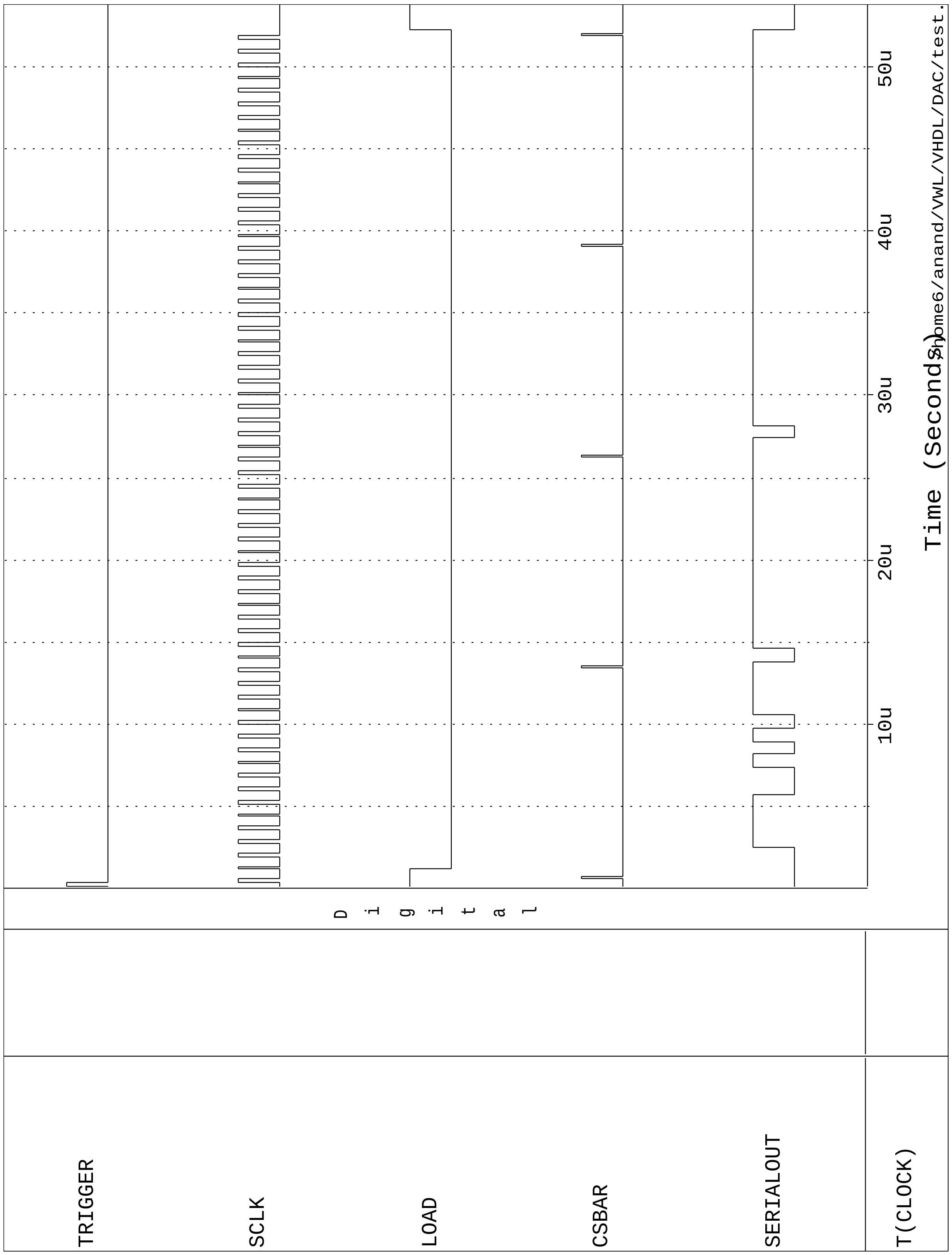


Figure 10. Waveforms for the DAC controller.

Figure 11 shows the pin-out of the XILINX FPGA configured as the DAC controller and Figure 12 is the LCA map of XILINX FPGA configured as the DAC controller.

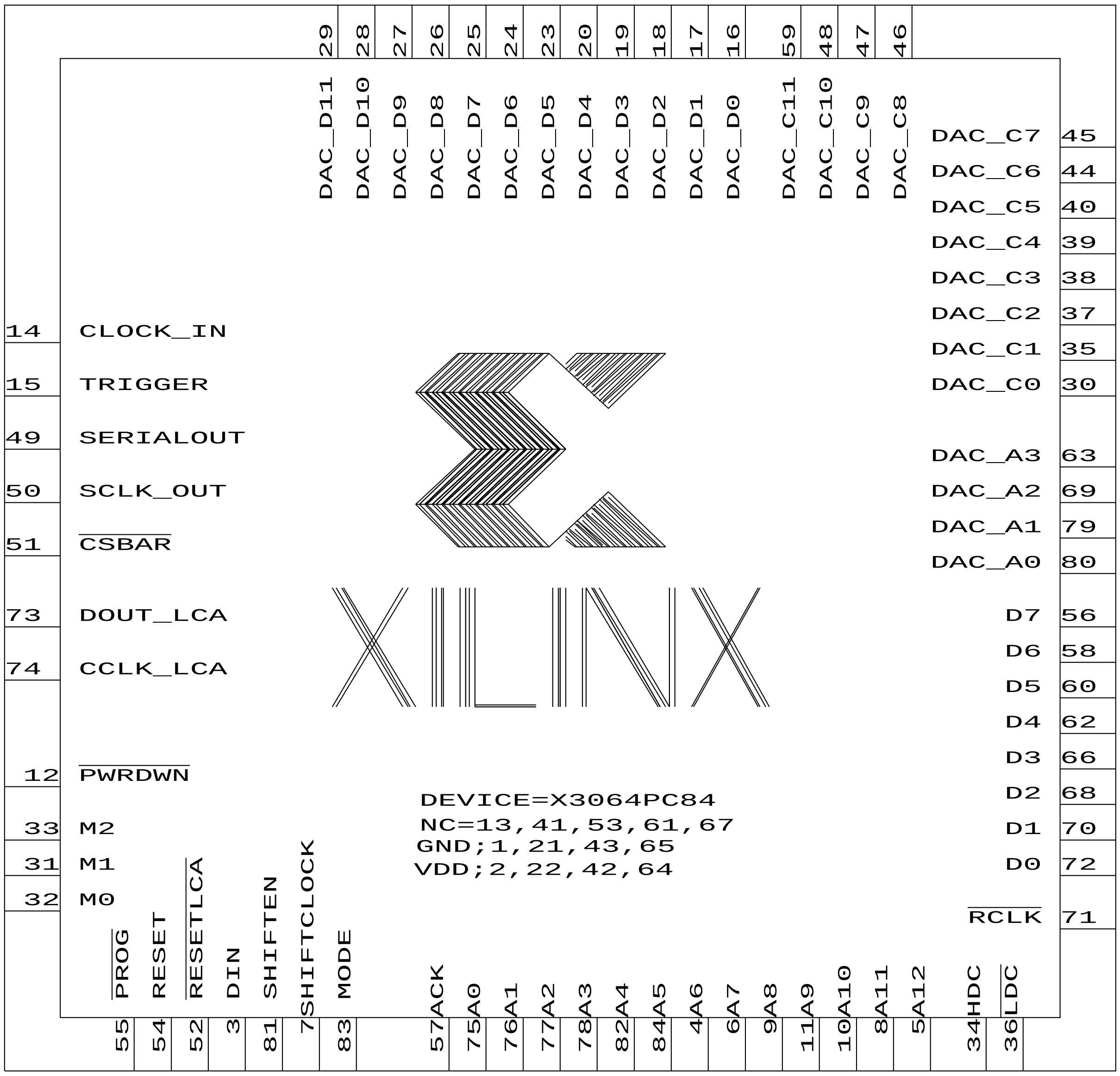


Figure 11. Pin-out for the DAC controller FPGA chip.

The DAC controller had to share pins along with the LCA's configuration pins used during the configuration cycle of the LCA device. External bus transceivers were used to multiplex the shared lines for the LCA configuration and the DAC controller. The external devices used were Quad 2 to 1 multiplexer (SN74LS157) for multiplexing the EPROM output and the hexadecimal switch outputs. Octal bi-directional bus transceivers (SN74ALS645A) were used for the Address bus for the EPROM and the hexadecimal switch outputs. The EPROM used is 27C64 (64Kbits). Detailed schematic of the SIC controller board is included in the appendix.

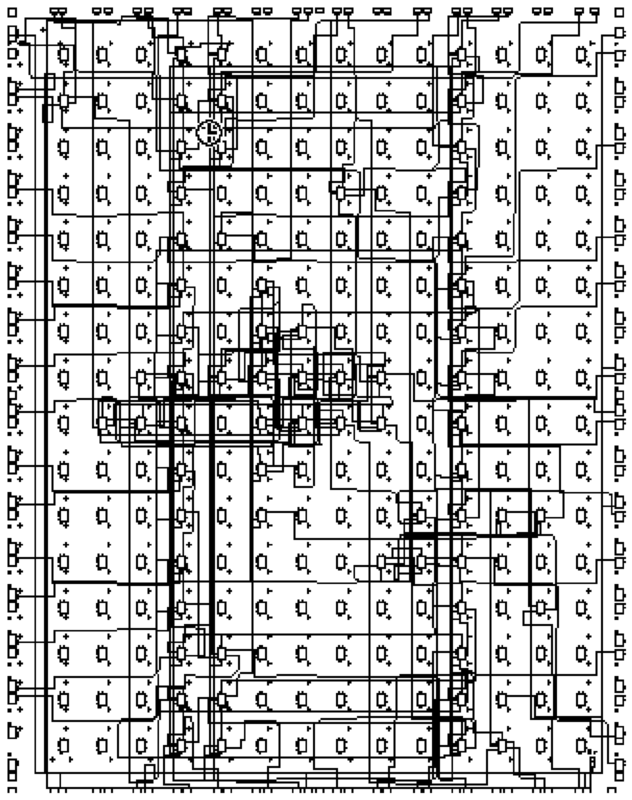


Figure 13. LCA map of the controller.

LCA DESIGN SUMMARY:
- Part type=3064PC84-70
- 66 of 224 CLBs used
- 67 of 70 I/O pins used
- 0 of 50 internal IOBs used
- 1 of 32 internal three-state signals used (1 TBUFS used)
- 82 CLB flip-flops used

Only 30% of the CLBs are used for the entire design. The design could have been targeted towards lower CLB FPGA's, but the pin count requirement was the factor influencing this decision.

5. Serial Interface Control in CMOS Shaper Integrated Circuit:

The Technique of Serial-Interface Control can be incorporated in to an CMOS shaper Integrated circuit. The CMOS shaper IC has the primary function of charge amplification and shaping of signals from detector components like cathode strips. In order to incorporate a digitally controlled stages for the gain and the shaping time *n* bits/function are needed for 2^n steps/function. This *n* bits/function figure is an unfavorable method as it increases the pin count and hence the package size and many unfavorable consequences like lead inductances following the package size increase. It is a good method to have serial interface controlled device technique for these Integrated circuits. Since the gain and shaping time are not going to be changing dynamically during the circuit operation, this technique can be implemented into an CMOS IC for the application discussed before. Figure 13 shows the functional diagram of an CMOS IC with SIC protocol.

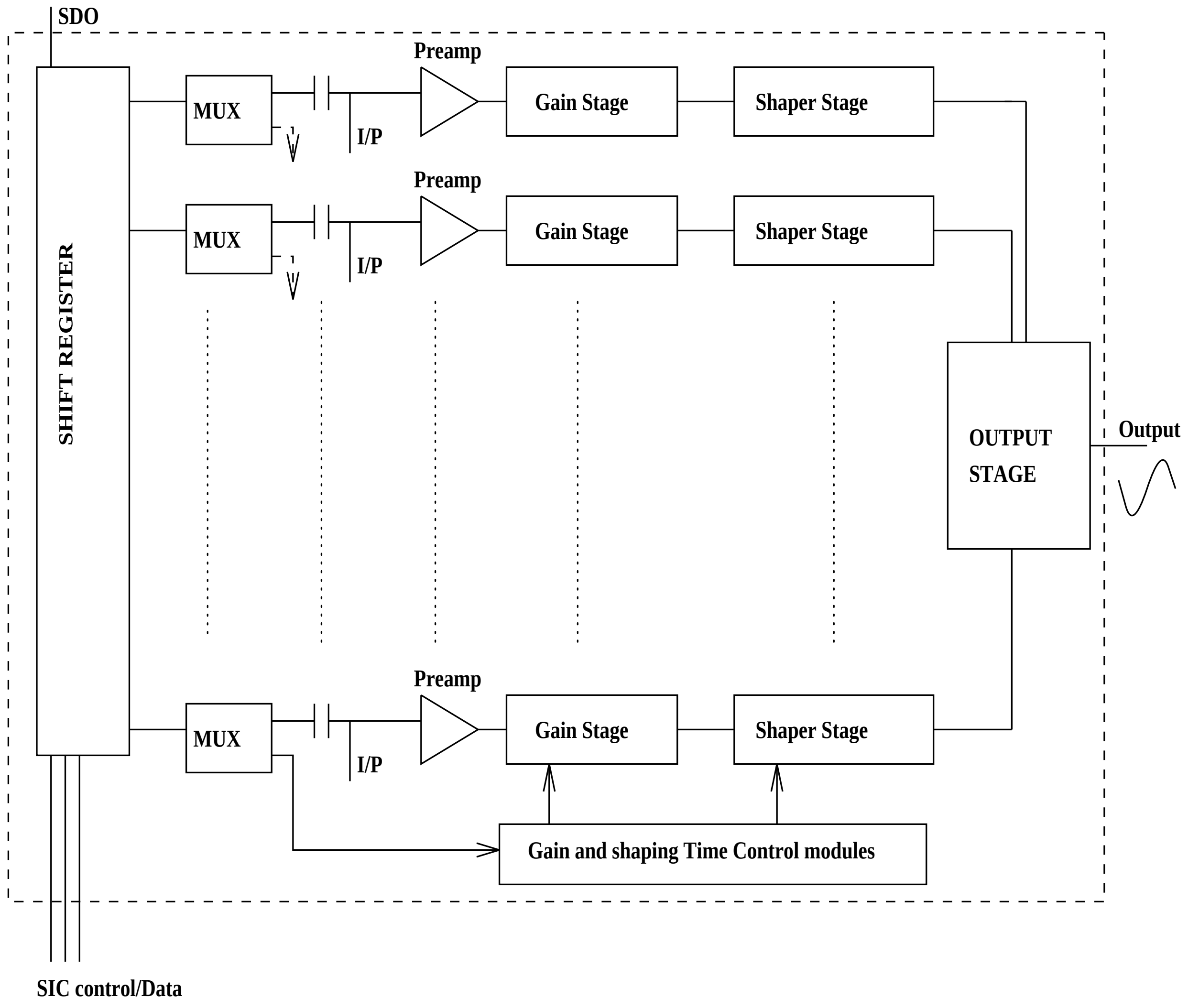


Figure 13. CMOS shaper Integrated Circuit with SIC.

The data bits pumped into the CMOS IC can be used to control the circuit parameters like gain/ shaping time and also in steering On chip precision capacitors for precise calibration of the pre-amplifiers. The data bits can also be used to control the mode of the timing discriminator (not shown in the figure), Leading Edge (LED) triggered or Constant Fraction (CFD). By making the CMOS shaper IC to be serial interface controlled and compatible with SPI/QSPI^{TM} or Microwire^{TM}, the advantages of lower pin count and compatibility with most low cost SPI/QSPI^{TM} or Microwire^{TM} supporting micro-controllers are enjoyed.

6. Testing and Results of the SIC:

The Serial Interface Controller was tested for its functionality using an logic analyzer to verify the stream of bits out of the download line, clock for the D2A and $\overline{CS}$ control signal. The Serial Interface Controller was realized by fabricating it as a motherboard which has provisions for housing a daughter board where the serially controlled device will reside. The schematics for the motherboard and the daughter board is given in Appendix A and the PCB layouts in Appendix C.

The results obtained from the testing of the entire SIC setup is shown as follows:

		OUTPUT in Volts			
	Vcontrib (V)	**DAC A**	**DAC B**	**DAC C**	**DAC D**
BIT 1	**1.25**	1.250341	1.250421	1.250004	1.250355
BIT 2	**625.0E-3**	625.35400E-3	625.44200E-3	624.95700E-3	625.31600E-3
BIT 3	**312.50E-3**	312.86400E-3	312.98000E-3	312.45000E-3	312.73400E-3
BIT 4	**156.250E-3**	156.61050E-3	156.70260E-3	156.09710E-3	156.45080E-3
BIT 5	**78.1250E-3**	78.51670E-3	78.58580E-3	77.96740E-3	78.32860E-3
BIT 6	**39.06250E-3**	39.43920E-3	39.52610E-3	38.91540E-3	39.24760E-3
BIT 7	**19.53125E-3**	19.89640E-3	19.99410E-3	19.38740E-3	19.71360E-3
BIT 8	**9.765625E-3**	10.14070E-3	10.23180E-3	9.61320E-3	9.95990E-3
BIT 9	**4.8828125E-3**	5.26580E-3	5.34470E-3	4.72880E-3	5.06520E-3
BIT 10	**2.4414063E-3**	2.82260E-3	2.91420E-3	2.28640E-3	2.61880E-3
BIT 11	**1.2207031E-3**	1.62230E-3	1.70180E-3	1.05440E-3	1.40680E-3
BIT 12	**610.35156E-6**	988.40000E-6	1.08640E-3	429.90000E-6	813.90000E-6

	Error in Volts			
	Error A	**Error B**	**Error C**	**Error D**
BIT 1	-341.0E-6	-421.0E-6	-4.0E-6	-355.0E-6
BIT 2	-354.0E-6	-442.0E-6	43.0E-6	-316.0E-6
BIT 3	-364.0E-6	-480.0E-6	50.0E-6	-234.0E-6
BIT 4	-360.5E-6	-452.6E-6	152.9E-6	-200.8E-6
BIT 5	-391.7E-6	-460.8E-6	157.6E-6	-203.6E-6
BIT 6	-376.7E-6	-463.6E-6	147.1E-6	-185.1E-6
BIT 7	-365.2E-6	-462.9E-6	143.9E-6	-182.4E-6
BIT 8	-375.1E-6	-466.2E-6	152.4E-6	-194.3E-6
BIT 9	-383.0E-6	-461.9E-6	154.0E-6	-182.4E-6
BIT 10	-381.2E-6	-472.8E-6	155.0E-6	-177.4E-6
BIT 11	-401.6E-6	-481.1E-6	166.3E-6	-186.1E-6
BIT 12	-378.0E-6	-476.0E-6	180.5E-6	-203.5E-6

Each bit is individually set for each DAC and the output is measured in the above case, and also results were verified for random DAC input settings. The following figure 14 shows the MAX 537 Error (Vout_actual - Vmeasured) along with the manufacturer listed $\pm$1 LSB total error.

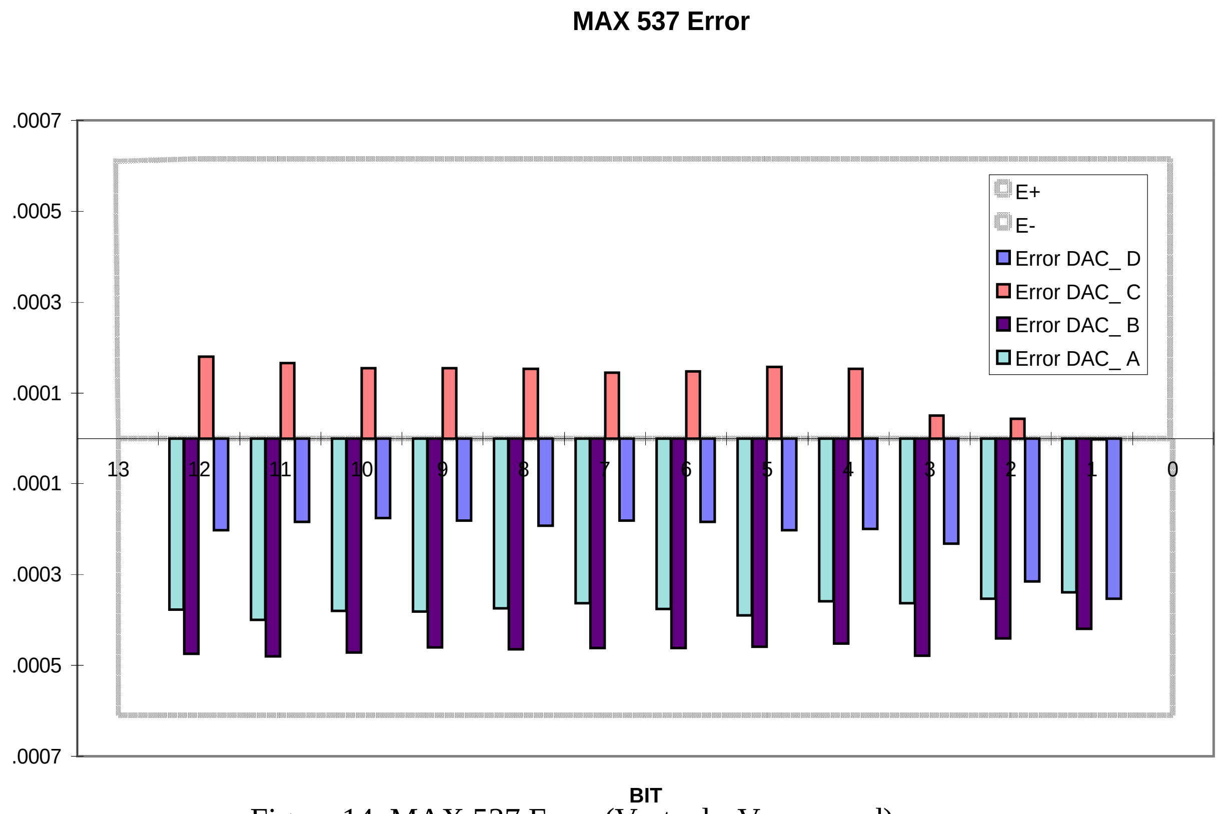


Figure 14. MAX 537 Error (Vactual - Vmeasured).

An user interface was written for MS Windows using Visual Basic. The program can be used to obtain the Digital code to be set for a given voltage and also to determine the output voltage for a particular code set. The control panel for the program is shown in Appendix D.

Appendix A. Schematics

List of figures:

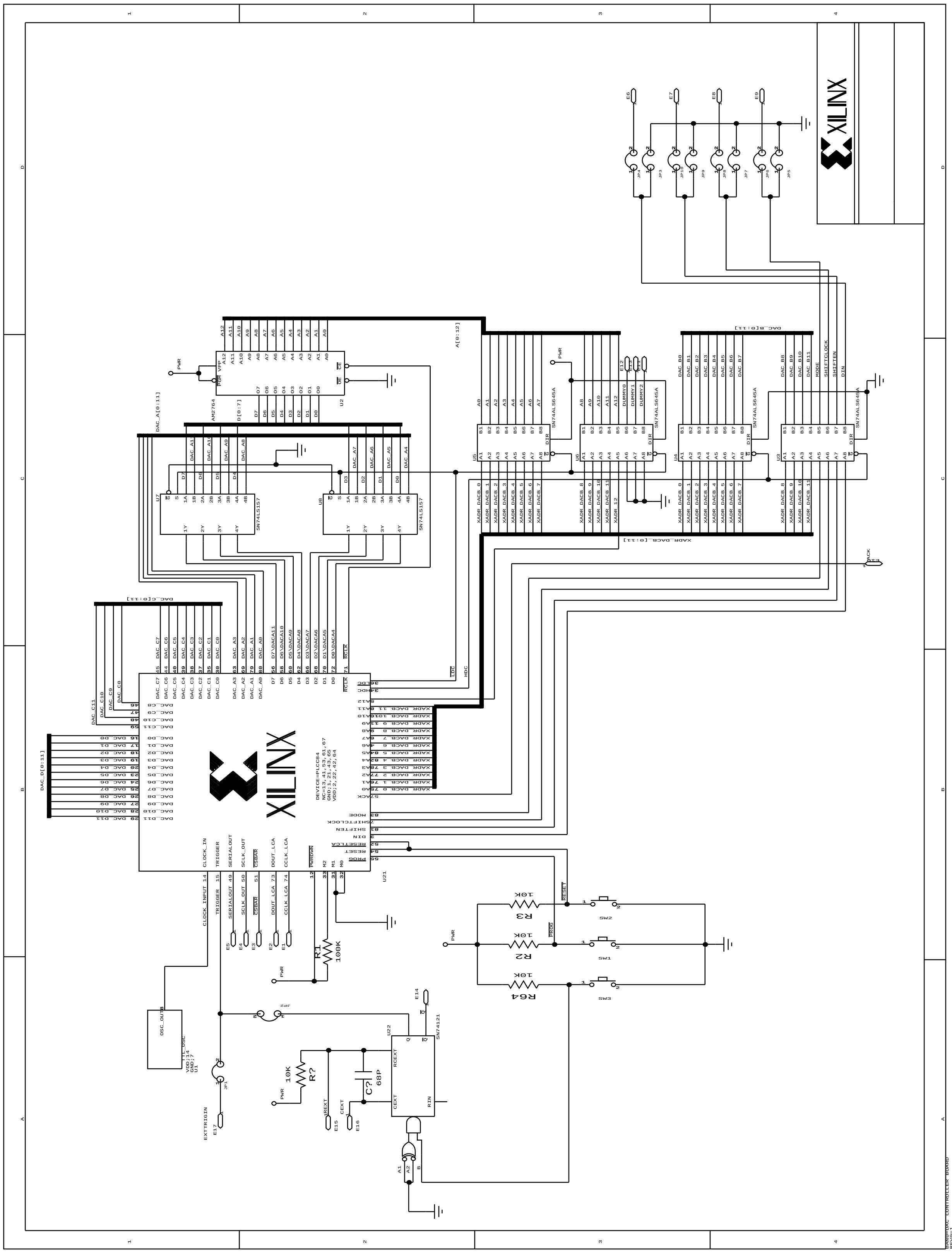


Figure A. Serial Interface Controller Board schematic (1/3).

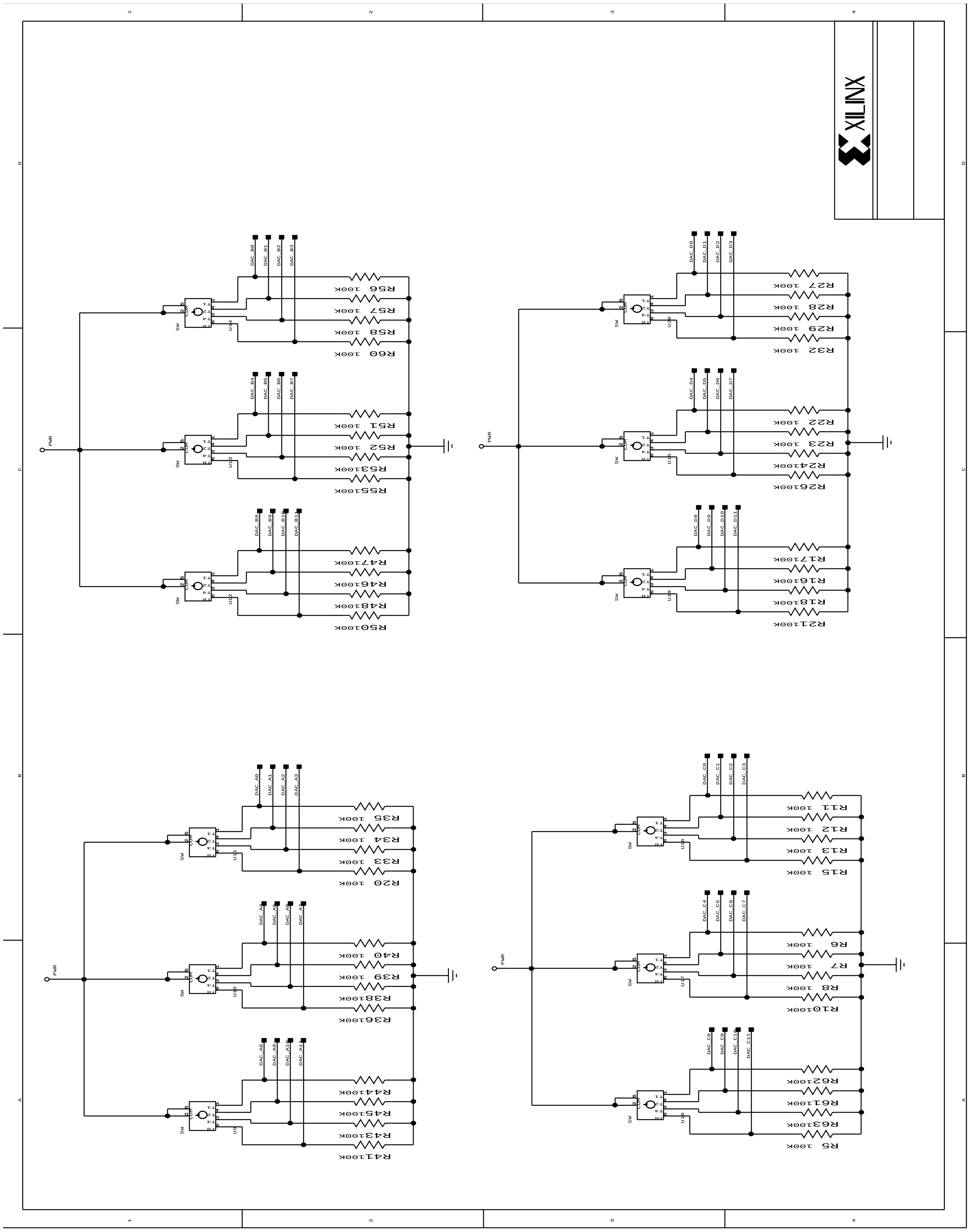


Figure B. Serial Interface Controller Board schematic (2/3).

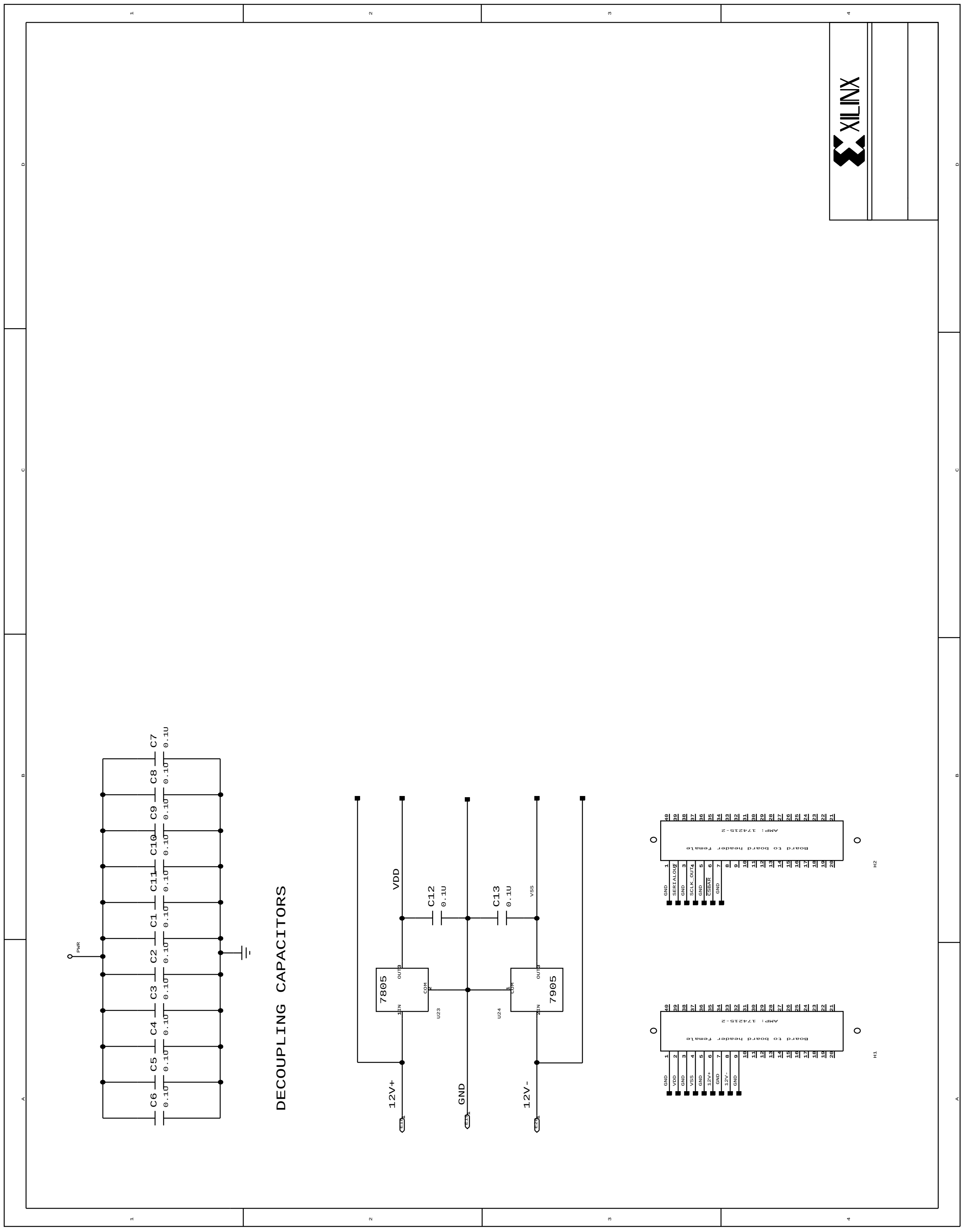


Figure C. Serial Interface Controller Board schematic (3/3).

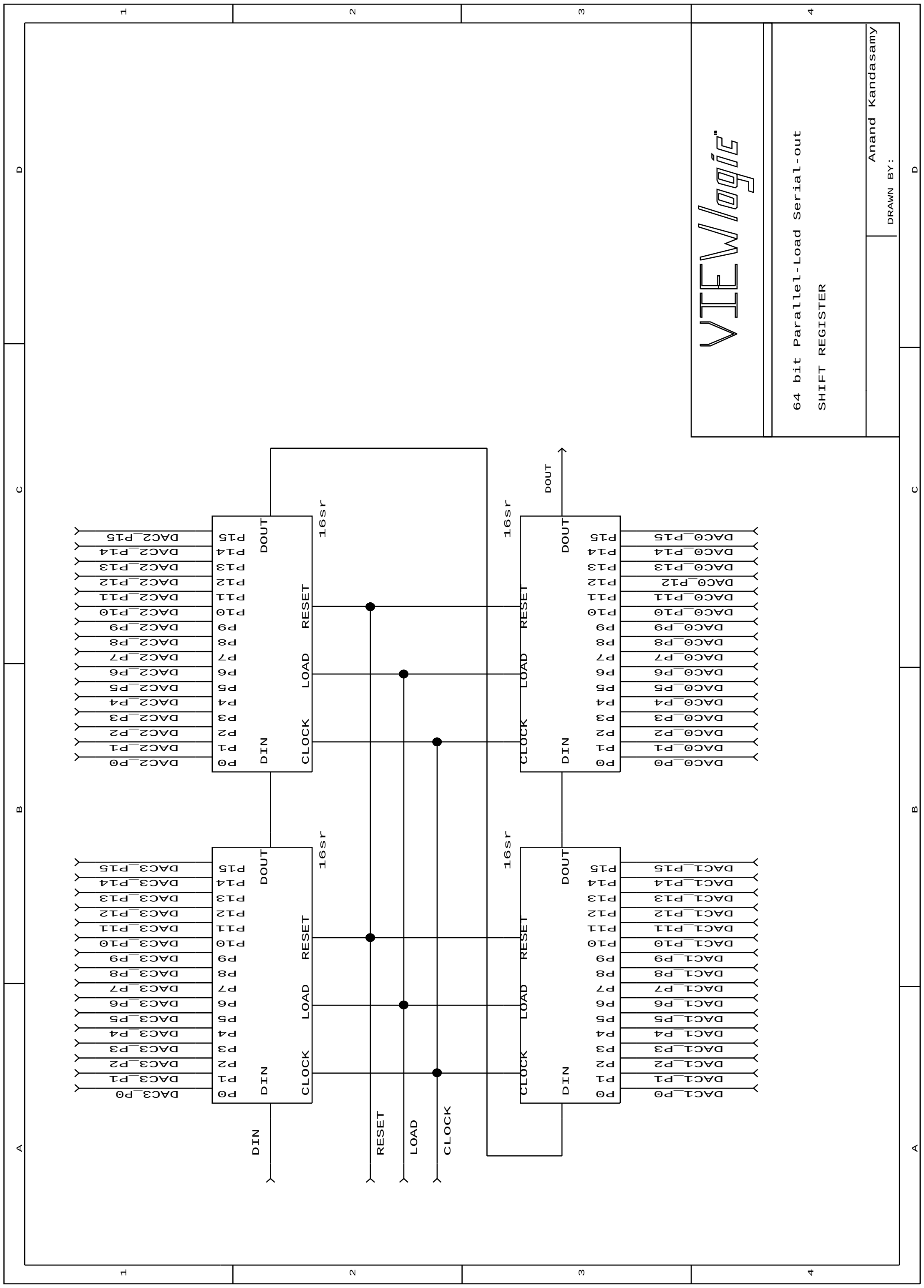


Figure D. 64 Bit shift register.

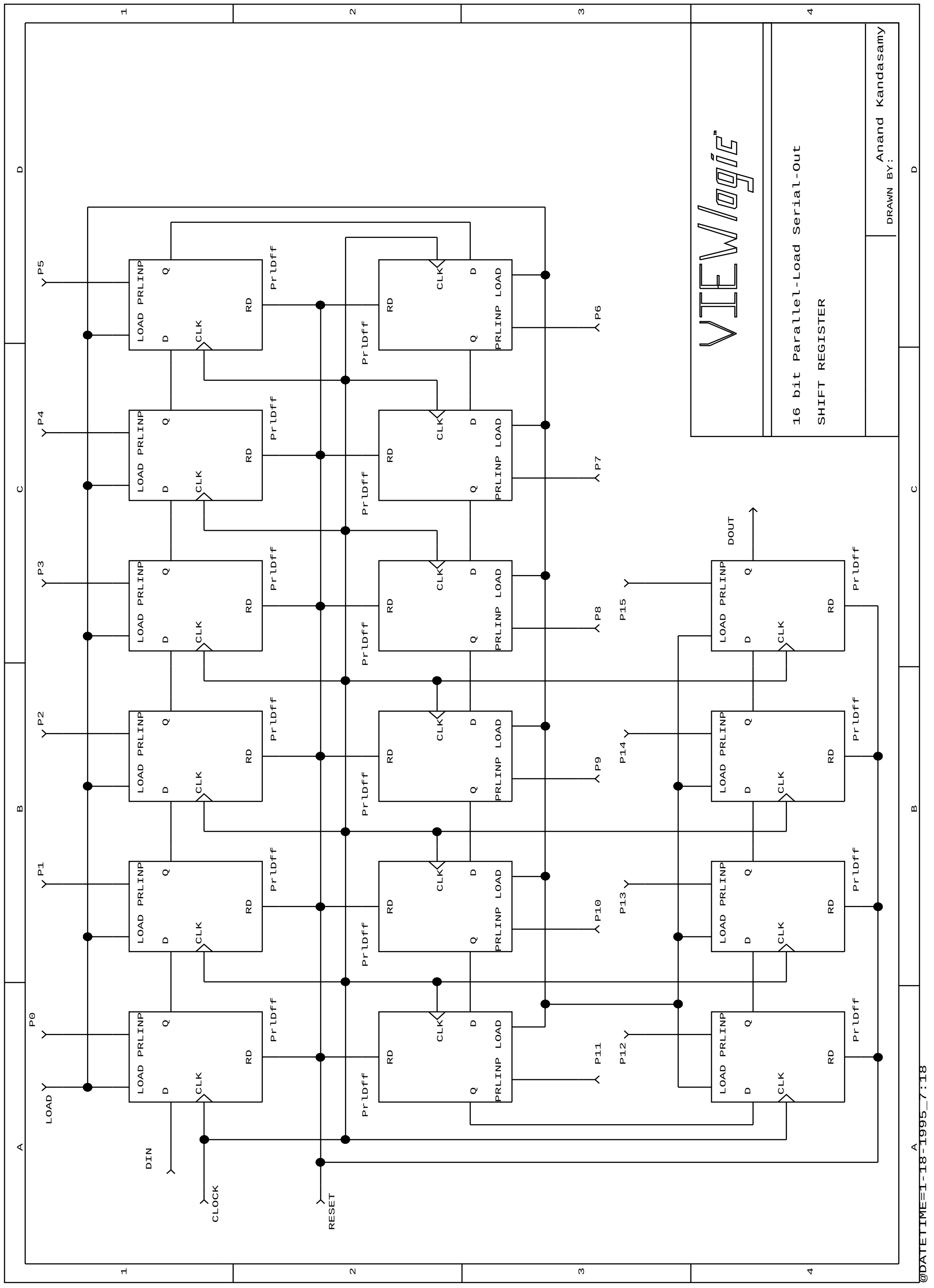


Figure E. 16 Bit parallel loadable shift register.

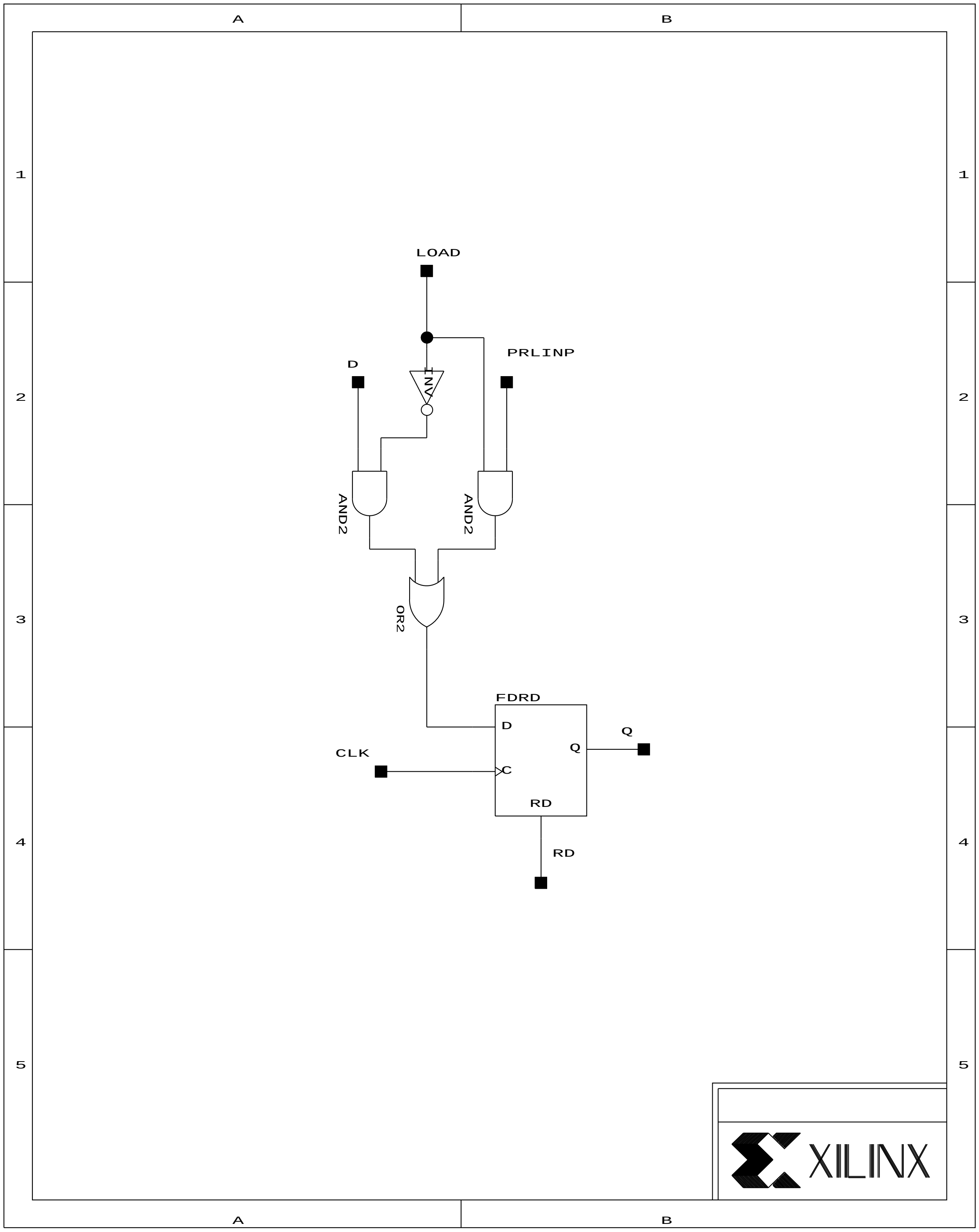


Figure F. 1 Bit Parallel load D-Flip-Flop.

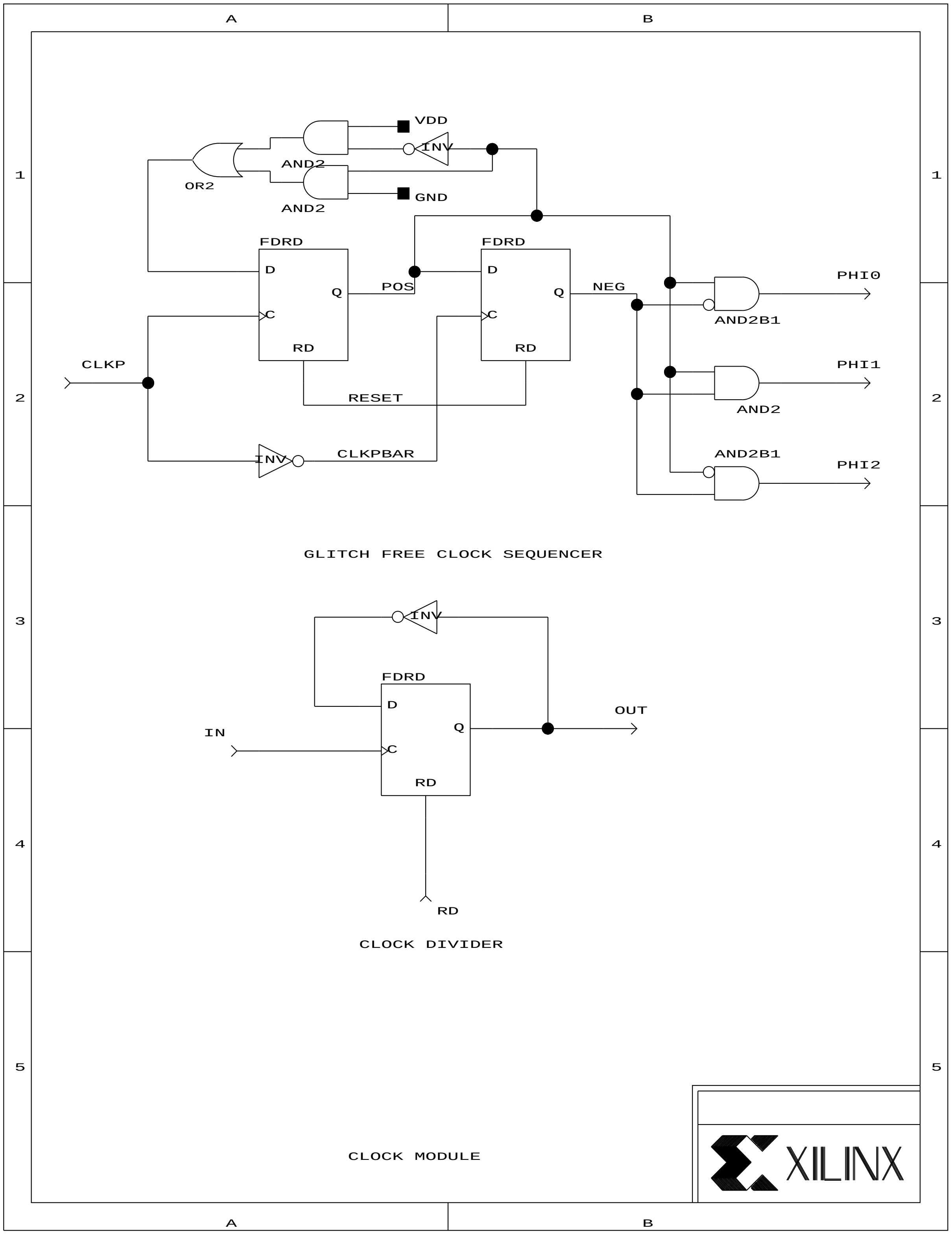


Figure G. Clock Module.

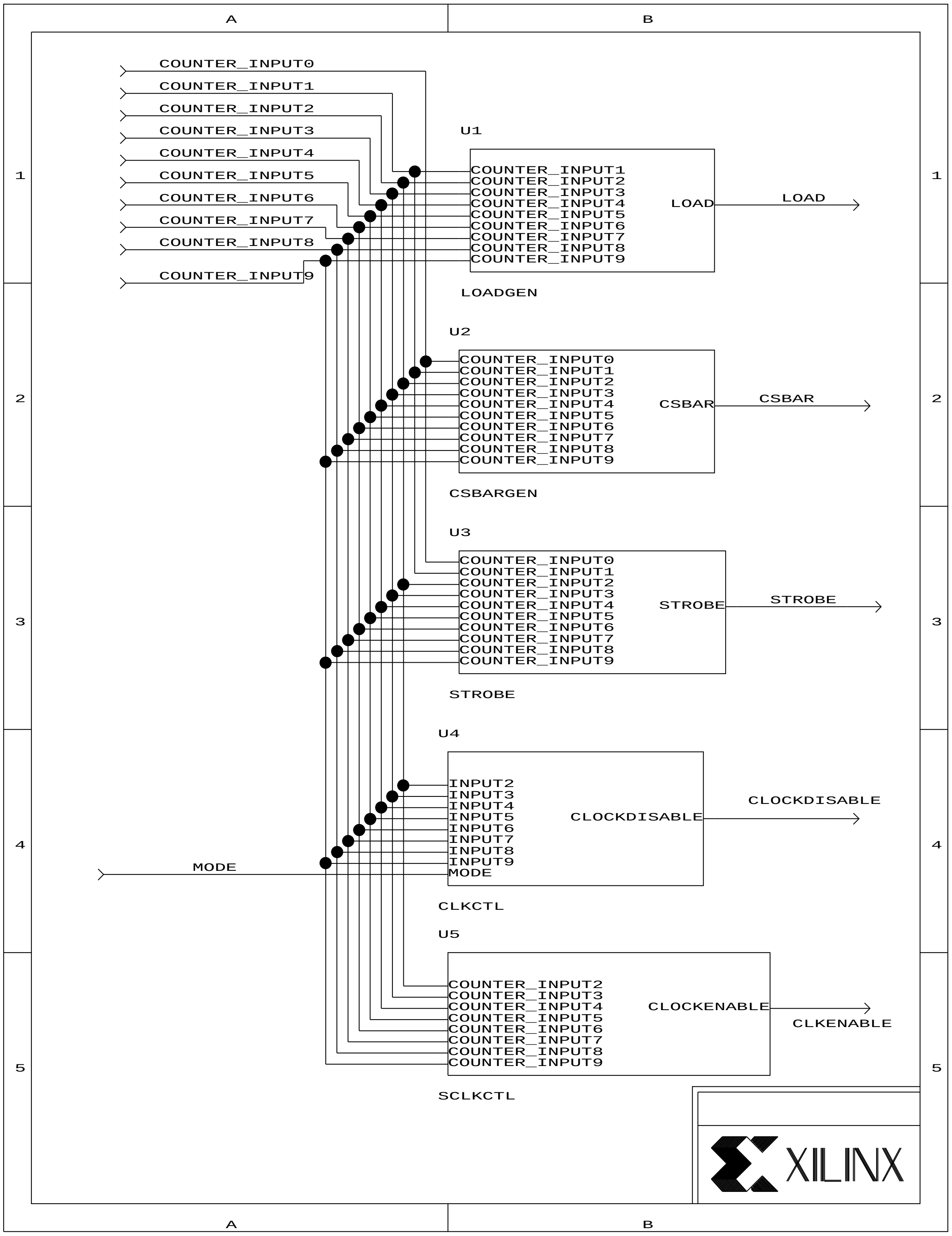


Figure H. Control Signals generating module.

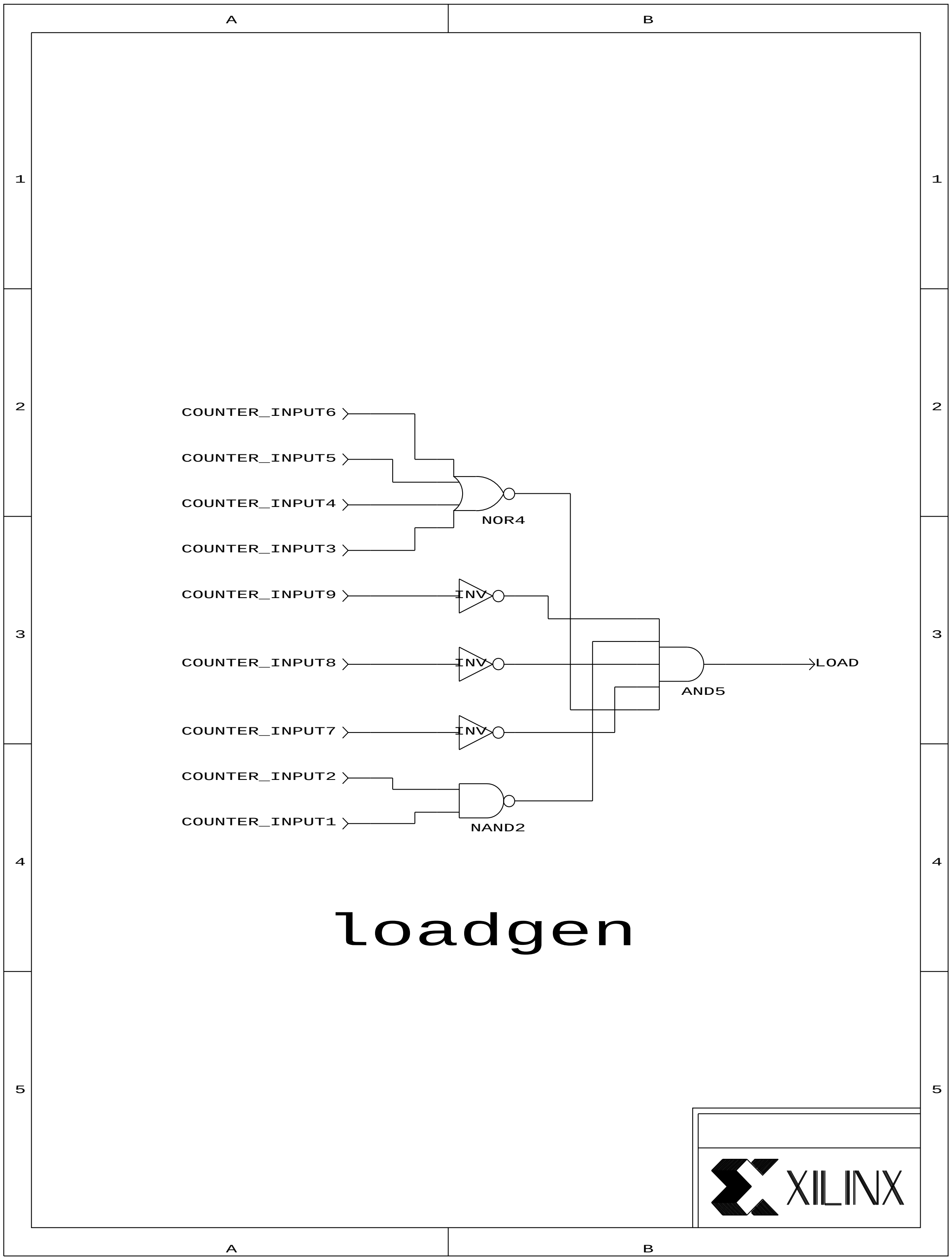


Figure I. LOAD signal generator.

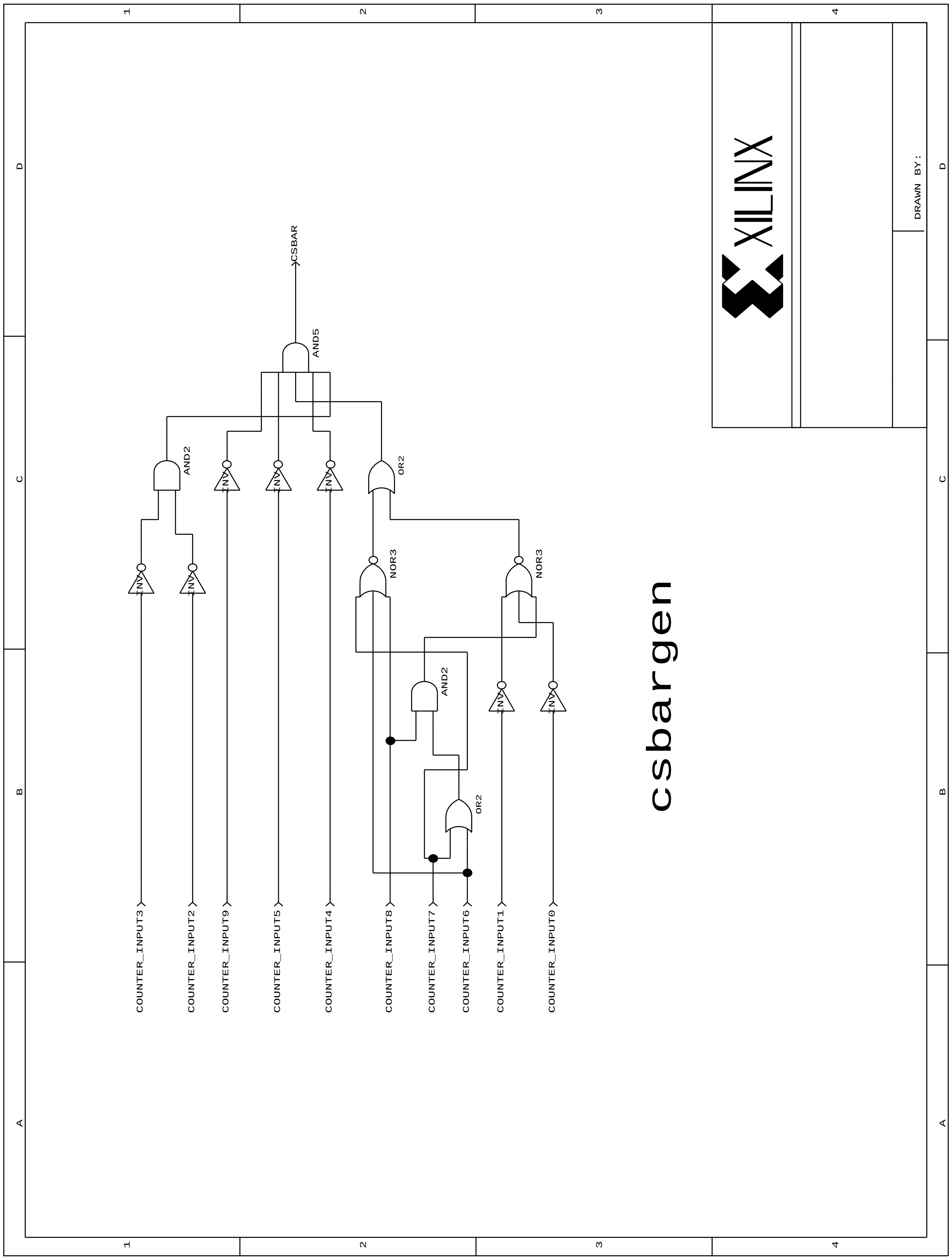


Figure J. $\overline{\text{CS}}$ signal generator.

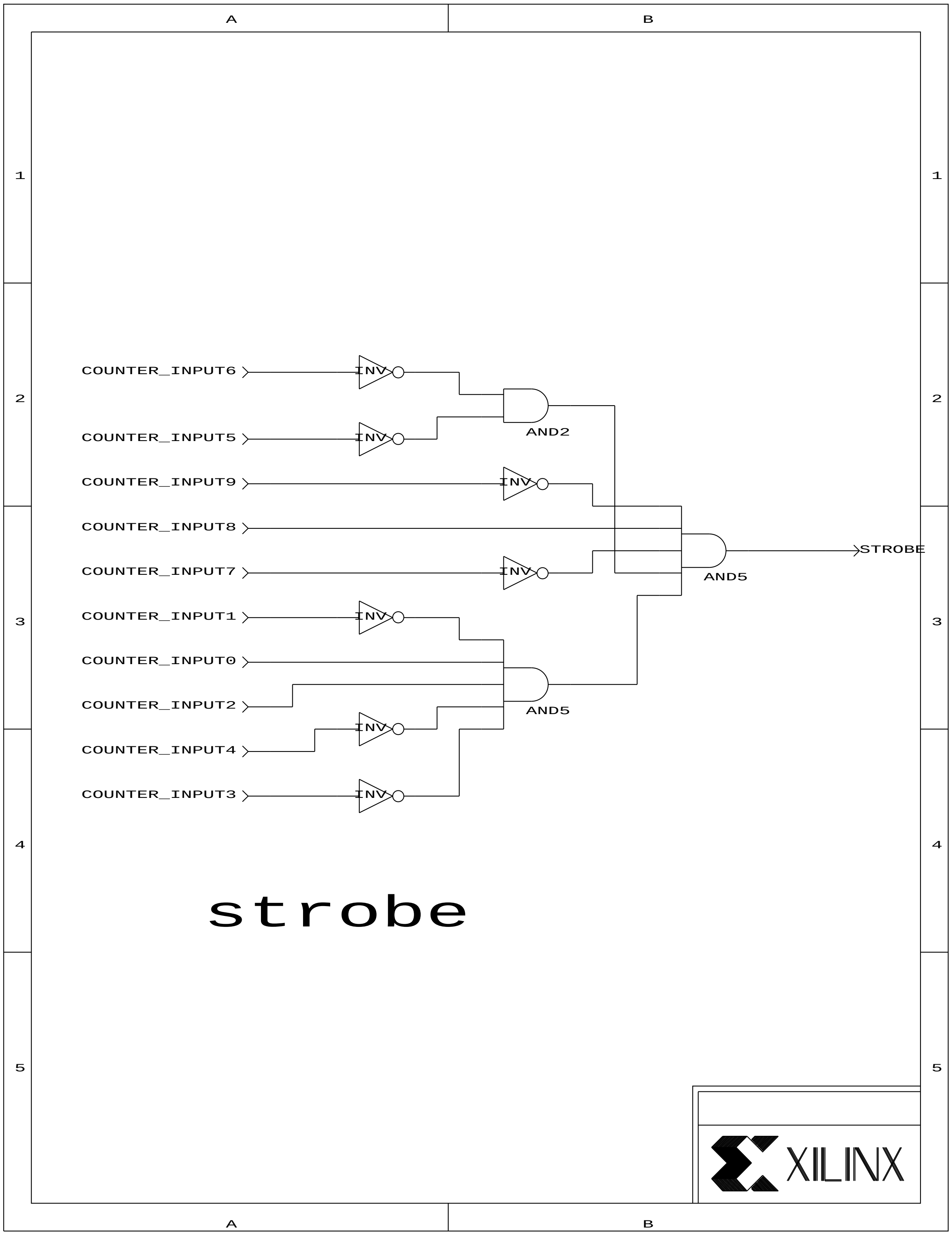


Figure K. Internal reset generator

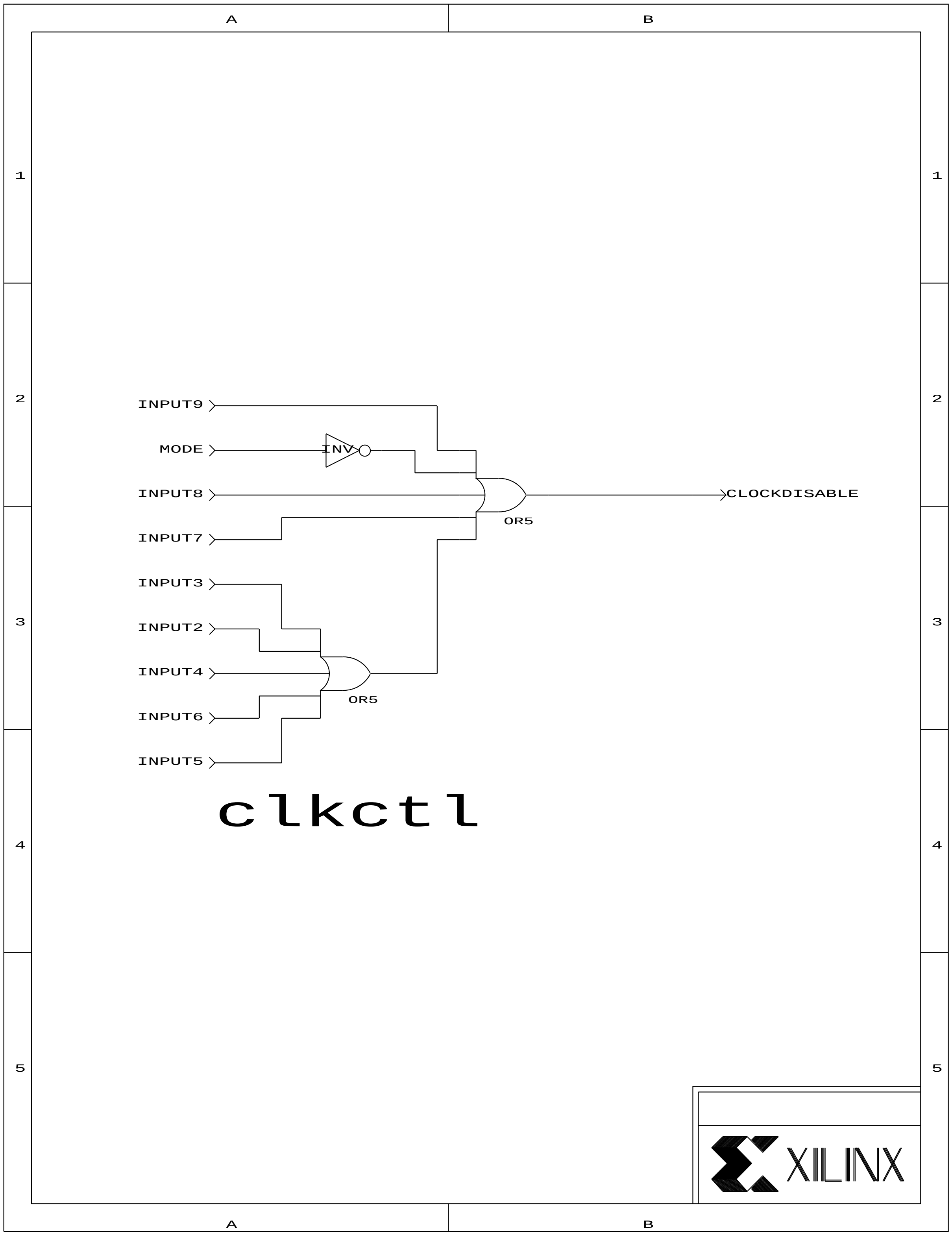


Figure L. Operating mode clock steering module.

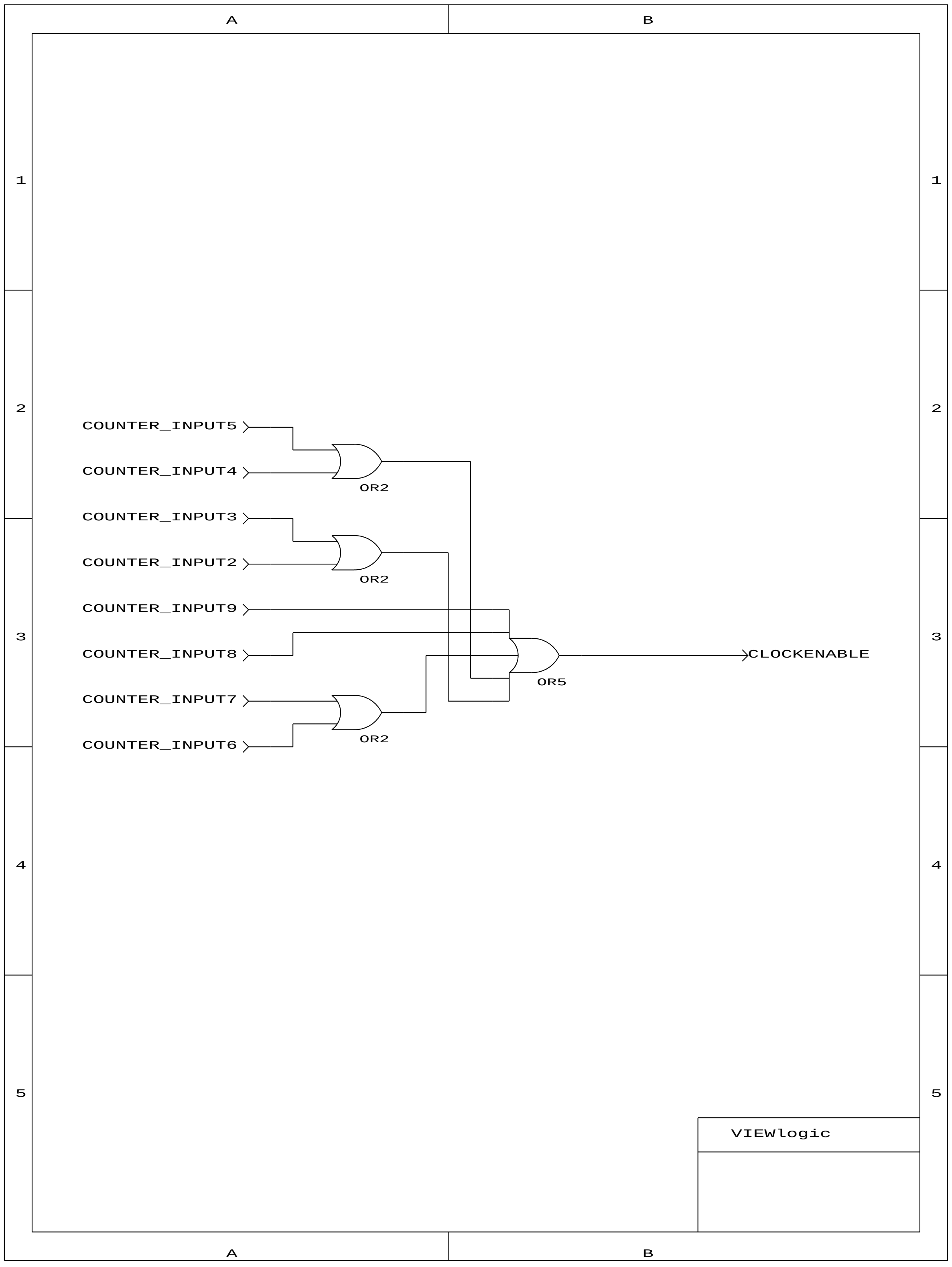


Figure M. SCLK enable control generator.

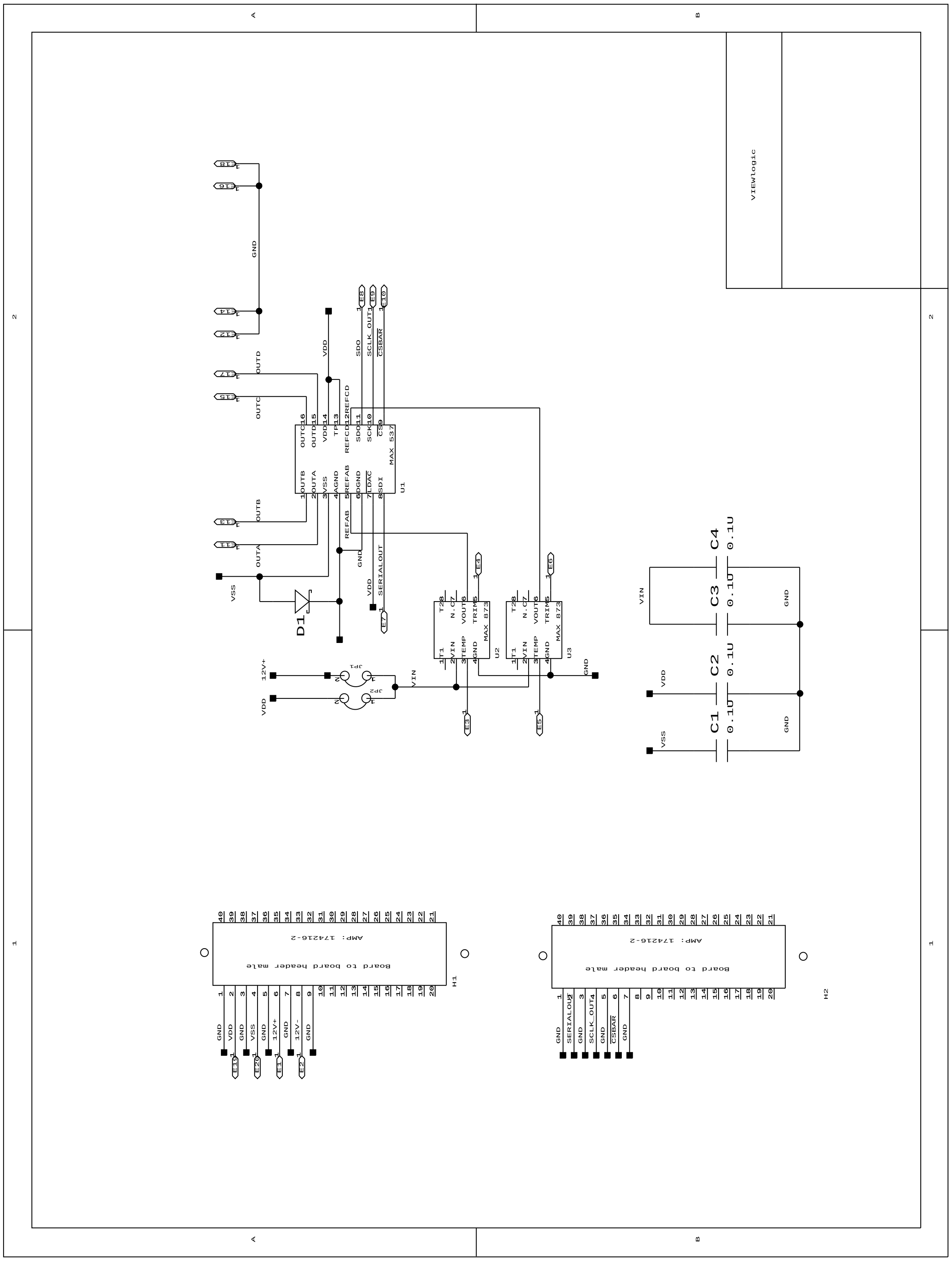


Figure N. D2A daughter board schematic.

Appendix B. VHDL Source Code

List of VHDL source codes:

1. Source code for LOAD signal generation.
2. Source code for Internal reset signal generation.
3. Source code for $\overline{\text{CS}}$ signal generation.
4. Source code for operating mode clock steering.
5. Source code for SCLK enable control.

All VHDL source code are written in behavior level description.

```
library synth;
use synth.stdsynth.all;
entity loadgen is
port (signal counter_input: in vlbit_1d(9 downto 0);
        signal load:      out vlbit);
end loadgen ;
-- Architecture body:
architecture behavior of loadgen is
begin
            control:process(counter_input)
                                    begin
                                    CASE (v1d2int(counter_input)) IS
                                    when 0 => load  <= ‘1’;
                                    when 1 => load  <= ‘1’;
                                    when 2 => load  <= ‘1’;
                                    when 3 => load  <= ‘1’;
                                    when 4 => load  <= ‘1’;
                                    when 5 => load  <= ‘1’;
                                    when others => load <=’0’;
                                    END CASE;
            end process;
end behavior;
```

Code 1. Source code for LOAD signal generation.

```
library synth;
use synth.stdsynth.all;
entity strobe is
port (signal counter_input: in vlbit_1d(9 downto 0);
        signal strobe :  out vlbit);
end strobe ;
-- Architecture body:
architecture behavior of strobe is
begin
            control:process(counter_input)
            begin
            CASE (v1d2int(counter_input)) IS
            when   0 => strobe <= ‘0’;
            when   7 => strobe <= ‘0’;
            when  15 => strobe <= ‘0’;
            when  31 => strobe <= ‘0’;
            when  63 => strobe <= ‘0’;
            when 127 => strobe <= ‘0’;
            when 255 => strobe <= ‘0’;
            when 260 => strobe <= ‘0’;
            when 261 => strobe <= ‘1’;
            when 262 => strobe <= ‘0’;
            when others => strobe <= ‘0’;
            END CASE;
            end process;
end behavior;
```

Code 2. Source code for Internal reset signal generation.

```
library synth;
use synth.stdsynth.all;
entity csbargen is
port (signal counter_input: in vlbit_1d(9 downto 0);
        signal csbar:      out vlbit);
end csbargen ;
-- Architecture body:
architecture behavior of csbargen is
begin
            control:process(counter_input)
            begin
            CASE (v1d2int(counter_input)) IS
            when   0 => csbar <= ‘1’;
            when   1 => csbar <= ‘1’;
            when   2 => csbar <= ‘1’;
            when   3 => csbar <= ‘1’;
            when   4 => csbar <= ‘0’;
            when  64 => csbar <= ‘0’;
            when  67 => csbar <= ‘1’;
            when  68 => csbar <= ‘0’;
            when 128 => csbar <= ‘0’;
            when 131 => csbar <= ‘1’;
            when 132 => csbar <= ‘0’;
            when 195 => csbar <= ‘1’;
            when 196 => csbar <= ‘0’;
            when 256 => csbar <= ‘0’;
            when 258 => csbar <= ‘0’;
            when 259 => csbar <= ‘1’;
            when 260 => csbar <= ‘0’;
            when others => csbar <= ‘0’;
            END CASE;
            end process;
end behavior;
```

Code 3. Source code for $\overline{\text{CS}}$ signal generation.

```
library synth;
use synth.stdsynth.all;
entity clkctl is
port (signal input : in vlbit_1d(10 downto 0);
        signal clockdisable:      out vlbit);
end clkctl ;
-- Architecture body:
architecture behavior of clkctl is
begin
            control:process(input)
            begin
            CASE (v1d2int(input)) IS
            when 1024 => clockdisable  <= ‘0’;
            when 1025 => clockdisable  <= ‘0’;
            when 1026 => clockdisable  <= ‘0’;
            when 1027 => clockdisable  <= ‘0’;
            when others => clockdisable <=’1’;
            END CASE;
            end process;
end behavior;
```

Code 4. Source code for operating mode clock steering.

```
library synth;
use synth.stdsynth.all;
entity sclkctl is
port (signal counter_input: in vlbit_1d(9 downto 0);
               signal clockenable:      out vlbit);
end sclkctl ;
-- Architecture body:
architecture behavior of sclkctl is
begin
            control:process(counter_input)
            begin
            CASE (v1d2int(counter_input)) IS
            when 0 => clockenable  <= ‘0’;
            when 1 => clockenable  <= ‘0’;
            when 2 => clockenable  <= ‘0’;
            when 3 => clockenable  <= ‘0’;
            when 4 => clockenable  <= ‘1’;
            when 5 => clockenable  <= ‘1’;
```

```
          when others => clockenable <='1';
          END CASE;

          end process;
end behavior;
```

Code 5. Source code for SCLK enable control.